LUCAS DE LUCA CASTRO

PROCEDIMENTOS DE MODELAGEM E UMA FERRAMENTA DE GERAÇÃO AUTOMÁTICA DE CÓDIGO

LAVRAS - MG

2010

LUCAS DE LUCA CASTRO

PROCEDIMENTOS DE MODELAGEM E UMA FERRAMENTA DE GERAÇÃO AUTOMÁTICA DE CÓDIGO

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação

Orientador:

Dr. André Vital Saúde

LAVRAS - MG
2010

LUCAS DE LUCA CASTRO

PROCEDIMENTOS DE MODELAGEM E UMA FERRAMENTA DE GERAÇÃO AUTOMÁTICA DE CÓDIGO

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação

APROVADA em ____ de _______________ de ______

Prof. Juliana Galvani Greghi – UFLA

Dr. André Vital Saúde

Orientador

LAVRAS- MG

2010

Ao meu pai, Sérgio,
À minha irmã, Gi
À minha namorada, Luíza,
Em especial, à minha mãe, Hilda.

AGRADECIMENTOS

A DEUS, que me iluminou ao longo desta jornada.

À minha mãe, por acreditar e lutar ao meu lado tornando tudo possível.

Ao meu pai, pela educação e honestidade que dele herdei.

À minha irmã, Gi, por iluminar os meus dias, mesmo que distante.

À minha madrinha Tereza e padrinho Kassen, pelo carinho e por me auxiliarem em cada passo de minha vida.

À minha família, pelas orações e pedidos de paz durante minha caminhada.

Ao Serjão, grande amigo e companheiro e ao Lucas de Oliveira, pela amizade, compreensão e vivência repleta de conselhos.

À Luiza, pela força e paciência.

Aos meus amigos Dyogo, Jhonatas, Ferreira, Rilson, Ednaldo, Gustavo, Hewerton, Claudim, Gustavão, Conrado, Julia, Flávia, Máira e Tuane.

À família Carramenha, pelo incentivo e apoio.

A todos da Mitah, por compartilhar o saber, em especial ao Ricardo Victório, pela experiência e conhecimento compartilhado.

Ao meu orientador, André Saúde, não somente pela orientação mas também por ter aberto janelas em minha vida acadêmica.

Ao Programa de Iniciação Científica e Tecnológica para Micro e Pequenas Empresas (BITEC), pela bolsa que me foi concedida viabilizando o estagio e me permitindo desenvolver parte deste trabalho.

Ao Programa Institucional Voluntário de Iniciação Científica da UFLA. (*PIVIC/UFLA*), do qual fiz parte.

A todos aqueles que contribuíram com apoio, dados e elementos para o desenvolvimento desta pesquisa e que aqui não foram nomeados, meus mais sinceros agradecimentos.

RESUMO

O mercado do mundo tecnológico contemporâneo demanda por produtos de alta qualidade em curtos espaços de tempo. Neste contexto, as empresas de desenvolvimento de software adotam padrões de projeto que garantem produtividade e manutenção em seus sistemas. Entretanto, são múltiplos os modelos, dificultando a criação de um modelo-padrão de desenvolvimento por estas empresas. Objetivou-se neste trabalho a criação de um gerador de código denominado BlueBox que otimiza a produção de empresas que utilizam o framework Iguassu como ferramenta de desenvolvimento de Software e a criação de práticas de modelagem de software, visando a geração dos códigos conforme o modelo Iguassu. Como resultados, descrevem-se o gerador CodeGen utilizando tecnologia XSLT, a utilização da tecnologia *Velocity* para a construção do gerador BlueBox e, por fim são listados os métodos para modelagem do diagrama de classes e de estados. Concluiu-se que o BlueBox tem maior rendimento em comparação ao CodeGen, tais ferramentas aceleram a produtividade das empresas que as adotam e é necessário a utilização dos procedimentos de modelagem para que a geração se dê de maneira correta.

Palavras-chave**:** geração de código, padrões de projeto, modelagem UML.

ABSTRACT

The market of technology in the contemporaneous world demands high-quality products in a short period of time. In this context, software development enterprises adopt design patterns that guarantee productivity and maintenance in their systems. However, there are multiple models, making it difficult for these enterprises to create a unique pattern of development. This work presents a new code-generator, named Bluebox, which optimizes the production for enterprises using the Iguassu development framework as a tool for software development and the creation of software modeling practices, seeking for the generation of codes according to the Iguassu model. As a result, it can be described the CodeGen generator, using XSLT technology, the use of Velocity technology to build the BlueBox generator and in the end are listed the methods to model the diagram of classes and states. It was concluded that BlueBox has higher performance comparing to CodeGen, these tools accelerate the use of modeling procedures in order to obtain a correct generation.

Keywords*: code generation; design pattern; UML modeling;*

LISTA DE ILUSTRAÇÕES

Figura 1 Processo de transfomação dos modelos MDA 18

Figura 2 Transformações baseadas em modelos 19

Figura 3 Modelo do gerador GreenBox 20

Figura 4 Exemplo de diagrama de classes 22

Figura 5 Estrutura da modelagem UML do exemplo 23

Figura 6 Modelo e pacotes, informações do modelo UML no arquivo XMI 24

Figura 7 Conteúdo do pacote 'exemplo' representado no arquivo XMI 24

Figura 8 Conteúdo da classe Pessoa: Atributos 'id', 'nome' e 'idade' 25

Figura 9 Identificação do tipo do atributo 'id' através de um identificador de referência 26

Figura 10 Destacado o identificador do atributo 'int' 27

Figura 11 Conteúdo de uma associação 27

Figura 12 Conteúdo da ponta de inicio da associação (Pessoa – Telefone) 29

Figura 13 Generalização no arquivo XMI, classe Aluno herda de classe Pessoa 30

Figura 14 Máquina de estados de Aluno 31

Figura 15 Conteúdo da Maquina de Estado de Aluno. 31

Figura 16 Conteúdo do elemento *UML:Region.subvertex*, os estados do diagrama. 32

Figura 17 Conteúdo do elemento UML: State com as referências das transições. 33

Figura 18 Informações das transições no arquivo XMI 33

Figura 19 Modelo MVC utilizado em aplicação web 34

Figura 20 Arquitetura Iguassu, com destaque na única camada que necessita implementação manual 36

Figura 21 Funcionamento do gerador 41

Figura 22 Estrutura do arquivo XML gerado 42

Figura 23 Exemplo XSL para declaração de uma classe Java 44
Figura 24 Chamada do *template* UML:Attribute para declarar os atributos...... 45
Figura 25 *Template* UML:Attribute para declaração de atributos 45
Figura 26 *Template* no escopo do tipo de atributo que imprime a declaração do atributo 46
Figura 27 Código gerado para a classe Pessoa, definida no *template* XSL....... 46
Figura 28 Arquitetura do BlueBox 48
Figura 29 Componentes do XMI Parse 49
Figura 30 de como obter dados de um elemento UML:Class pelo JColtrane.... 50
Figura 31 Declaração de pacote 55
Figura 32 Iteração para cada inteface e verificando se existe a interface ActivityInstance. 55
Figura 33 Verificando se a classe possui super classe 55
Figura 34 Listando atributos primitivos 56
Figura 35 Listando atributos associados... 56
Figura 36 Listando todos os atributos 56
Figura 37 Listando marcadores de uma classe.. 57
Figura 38 Listando marcadores dos atributos associados de uma classe........... 57
Figura 39 Exemplo de modelagem dos atributos dentro de uma classe 59
Figura 40 Entidade Aluno possui realização com StateMachineEntity............. 60
Figura 41 Representação do atributo ‘telefones’ da classe Pessoa em uma associação 61
Figura 42 Criação da classe TipoTelefone, não devendo criar mais de uma associação entre classes 62
Figura 43 Utilização de validadores no atributo 63
Figura 44 Estrutura de pacotes de acordo com os procedimentos 64
Figura 45 Criação da máquina de estados da entidade Aluno 65
Figura 46 Máquina de estados da entidade Aluno 66

LISTA DE TABELAS

Tabela 1Valores e características de um término da associação....................... 28

LISTA DE ABREVIATURAS

CIM **-** *Computation Independent Model*

CRUD – *Create, Read, Update e Delete*

ER – Entidade Relacionamento

HTTP – *Hyper Text Transfer Protocol*

JAXB – *Java Architecture for XML Binding*

MVC –*Model/View/Control*

MDA – *Model Driven Architecture*

OMG – *Object Management Group*

PIM – *Plataform Independent Model*

PSM – *Plataform Specific Models*

SOA – *Service-Oriented Architecture*

TI – Tecnologia da Informação

UML – *Unified Modeling Language*

W3C – *World Wide Web Consortium*

XML – *eXtensible Markup Language*

XMI – *XML Metadata Interchange*

XSL – *eXtensibleStylesheet Language*

XSLT –*eXtensibleStylesheetLanguage Transformations*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 Objetivos 15
1.2 Estrutura do Trabalho 15
2 REFERENCIAL TEÓRICO 17
2.1 Model Driven Architecture (MDA) 17
2.2 Gerador Automático de Código 19
2.2.1 Arquivo XMI 20
2.3 Iguassu Framework 34
3 MATERIAL E MÉTODOS 38
3.1 Tipo de Pesquisa 38
3.2 Método 38
4 RESULTADOS 40
4.1 CodeGen 40
4.1.1 Arquitetura 40
4.2 BlueBox 47
4.2.1 Arquitetura 47
4.3 Procedimentos de modelagem 58
4.3.1 Diagrama de Classes 58
4.3.2 Diagrama de Estados 64
5 CONCLUSÕES 68
5.1 Trabalhos Futuros 69
6 REFERÊNCIAS 70

1 INTRODUÇÃO

Tendo-se em vista o atual conceito de produção, seja ela de mercadorias ou idéias, observa-se uma demanda do mercado em obtenção de produtos de alta qualidade em mínimos espaços de tempo. Na área de tecnologias de informação, a busca pela aceleração da elaboração de resultados e produtos, inerentes às necessidades contemporâneas, cresce cada vez mais, visando atingir os objetivos de cada empresa pertencente ao universo capitalista.

Neste contexto, observa-se que clientes de empresas de desenvolvimento de software têm exigido sistemas com alta qualidade e rápida produção. Assim, estas empresas enfrentam duas necessidades a serem atendidas na elaboração do ciclo de vida de um projeto: produtividade e manutenção. A fim de ampliar a produtividade e simplificar a manutenção de seus desenvolvedores, os arquitetos de software adotam padrões de projetos visando o auxílio no desenvolvimento de seus produtos.

Especificamente para sistemas de web, que muito tem crescido nos últimos anos, há uma diversidade de frameworks, sendo os mais utilizados aqueles baseados no padrão MVC (*Model/View/Control*). Entretanto, apesar de os mesmos auxiliarem no desenvolvimento, principalmente,da camada de comunicação web, estes deixam os engenheiros de software livres para propor qualquer arquitetura de desenvolvimento. Desta forma, estes arquitetos necessitam de tempo hábil e, muitas vezes, apresentam dificuldades para relacionar os frameworks e criar um modelo de desenvolvimento dos mesmos. Acredita-se, então, que, com um modelo, seria possível convergir os padrões de geração de código, acelerando a produtividade das empresas e, assim, garantindo a satisfação dos clientes.

Tendo em vista uma redução no tempo dispensado para desenvolvimento de arquiteturas de software, foi proposta a arquitetura Iguassu,

que permite o desenvolvimento de softwares web combinados com frameworks MVC.

A arquitetura Iguassu combina padrões de projeto e organiza o código do sistema tendo em vista as seguintes características inerentes aos mesmos: (1) o comportamento comum das classes no software e(2) a padronização do procedimento de cada tipo de classe. Aquele comportamento comum permite a criação de componentes reutilizáveis de software, bem como esta implementação padronizada permite a geração automática de código. Essas características aceleram o desenvolvimento do software e facilitam sua manutenção à posteriori, filosofia na qual se idealiza a Iguassu.

O framework Iguassu é uma implementação Java da arquitetura Iguassu. Este tem padrões de implementação pré-estabelecidos, sendo eles baseados em informações que podem ser encontradas nomodelo UML (*Unified Modeling Language*). Tal framework, entretanto, ainda não é um produto comercial, já que ainda necessita de medidas evolutivas, sendo uma destas evoluções a criação de geradores automáticos de código, objeto de estudo neste projeto. Para tanto, é preciso que haja um conhecimento específico em modelagem de software, tendo em vista a necessidade de um diagrama conter os recursos a serem utilizados pelo gerador na criação do código automático.

Referindo-se, especificamente, a este modelo de geradores de código automático, foram criados os produtos BlueBox, que utiliza a tecnologia *Velocity* e o gerador CodeGen, utilizando a tecnologia XSLT*(eXtensible Stylesheet Language Transformations)*. O BlueBox mostra-se com rendimento superior ao CodeGen, pois sua utilização se dá de maneira simplificada, facilitando o seu aprendizado e, desta forma, também de forma acelerada, atingindo mais rapidamente os resultados objetivados.

Os dados, entretanto, para serem utilizados pelo BlueBox, devem ter sido modelados anteriormente, seguindo, neste caso, os procedimentos de modelagem do Iguassu, que serão mais bem definidos ao longo do trabalho.

1.1 Objetivos

O objetivo geral deste trabalho foi a criação de um gerador de código denominado BlueBox que otimiza as produções de softwares que utilizam o framework Iguassu como ferramenta de desenvolvimento e a criação de práticas de modelagem, visando a geração dos códigos conforme o modelo Iguassu.

Tem-se como objetivos específicos:

- ✓ Estudo da metodologia MDA (*Model Driven Architecture*)
- ✓ Estudo da modelagem UML
- ✓ Verificação do impacto da modelagem UML na geração de arquivo de dados XMI (*XML Metadata Interchange)*
- ✓ Proposta de procedimento de modelagem UML adequado ao gerador de código
- ✓ Estudo do comportamento das classes do framework Iguassu
- ✓ Estudo da tecnologia XSLT
- ✓ Estudo da tecnologia *Velocity*
- ✓ Proposta de um gerador automático de código
- ✓ Criação de *templates* para a geração de código no formato do framework Iguassu

1.2 Estrutura do Trabalho

A primeira Seção introduz o trabalho, contextualizando e apresentando os objetivos do mesmo. A Seção 2 apresenta o referencial teórico, com uma pequena revisão de trabalhos realizados na literatura a respeito do Model Driven Architecture (MDA), Gerador automático de código, arquivo XMI e Iguassu

Framework. Na Seção 3 são descritos os materiais e métodos do trabalho, apresentando também o tipo de pesquisa utilizada. A Seção4 apresenta os resultados obtidos, dividindo-os em três subseções: Gerador XSL (*eXtensible Stylesheet Language)*, BlueBox e Procedimentos de Modelagem. Na primeira subseção descreve-se o gerador automático de código utilizando tecnologia XSL (CodeGen). Em seguida define a utilização da tecnologia *Velocity* para a construção do gerador BlueBox. Por fim são listados os métodos para modelagem do diagrama de classes e diagrama de estados. Na Seção 5são apresentadas as conclusões a respeito do que foi descrito na monografia e na Seção 6 são apresentadas as referencias utilizadas ao longo da elaboração do trabalho.

2 REFERENCIAL TEÓRICO

A presente seção discorrerá a respeito do embasamento teórico utilizado para a execução deste trabalho. A Seção 2.1 descreve a metodologia Model Driven Architecture (MDA) que objetiva descrever a utilização de modelos e como estes são preparados e relacionados entre si. O Item 2.2 expõe o funcionamento de um gerador automático de código genérico, e apresentação dos dados de um modelo UML em um arquivo XMI. Por fim, A Seção 2.3 apresenta a arquitetura do framework Iguassu, destacando as camadas nas quais os códigos podem ser gerados.

2.1 Model Driven Architecture (MDA)

Com a evolução da engenharia de software, o desenvolvimento de sistemas obteve avanços tanto no projeto, quanto na implementação de software.

O aumento no nível de abstração de software é um dos principais avanços, já que, antes, nas implementações eram utilizadas linguagens de baixo nível (linguagens próximas as linguagem de máquina) que dificultavam o entendimento e manutenção do código. Com a necessidade das empresas em desenvolver sistemas complexos e obter alta produtividade de software, estas adotaram novas linguagens e paradigmas de programação com alto nível de abstração (MELLOR et al, 2004).

Mellor *et al*, 2004, define Model Driven Architecture(MDA) um novo paradigma que utiliza modelos como centro do desenvolvimento do sistema, através destes o código pode ser gerado.

O MDA é uma metodologia reconhecida pela OMG(*Object Management Group*), para desenvolvimento de software destacando a importância do processo de modelagem de software, sendo possível a transformação de modelos abstratos em modelos mais concretos. Esse processo

de transformação tem como objetivo alcançar a geração do código fonte de maneira automatizada.

A MDA na produção de software define que devem ser criados três modelos descritos a seguir.

Computation Independent Model (CIM): é um modelo que apresenta uma visão do sistema independente da visão computacional, apresentando apenas os requisitos do sistema, informações do funcionamento do sistema que identificam o que realmente o sistema deve fazer.

Plataform Independent Model (PIM): é um modelo que não depende da tecnologia que será utilizada, apresentando o comportamento do software em alto nível de abstração.

Plataform Specific Models (PSM): este modelo é gerado considerando-se as especificações de determinada tecnologia que será utilizada durante o processo de implementação.

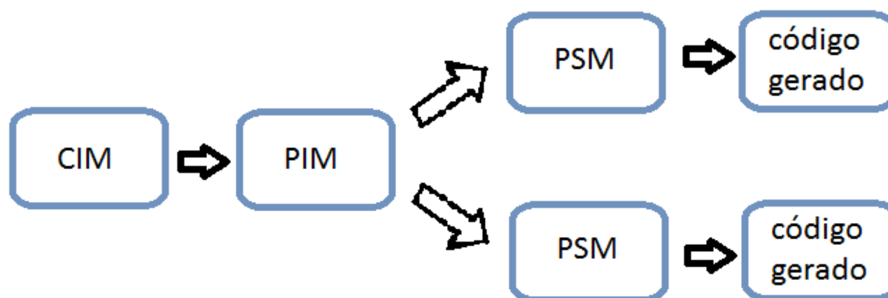


Figura 1 Processo de transformação dos modelos MDA

A Figura 1 mostra a transformação dos modelos MDA, do mais abstrato (CIM) até o mais concreto (PSM). Para ocorrer a transformação de um modelo em outro é utilizado um conjunto de regras especificas que descrevem tal processo.

2.2 Gerador Automático de Código

Um gerador automático de código tem como princípio a utilização de uma base de dados e *templates*. A base de dados é um arquivo que contem todas as informações que são utilizadas durante a geração de código. Os *templates*, por sua vez, definem como serão apresentados os dados no arquivo de saída. Estas informações são, então, processadas pelo gerador de código, computando a saída (código gerado). Caso haja a necessidade de alterações no código de saída, as transformações podem ser realizadas diretamente nos *templates*, sem que haja modificações no gerador. Isto possibilita que o código gerado adeque-se aos padrões/formatos que o Framework necessita.

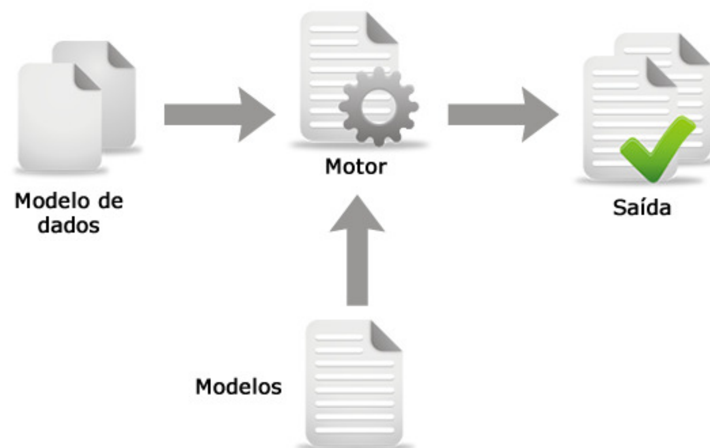


Figura 2 Transformações baseadas em modelos

Fonte BARBOSA (2009)

Uma das bases de dados que o gerador pode utilizar é o arquivo XMI (*XML Metadata Interchange*) como é utilizado pelo gerador GreenBox (2007), um gerador para desenvolvimento de aplicações em diversas tecnologias, independente da arquitetura ou padrões utilizados.

O GreenBox utiliza os *templates Velocity* para definir como serão apresentadas as informações no arquivo de saída como representado na Figura 3. Esta é uma aplicação do diagrama genérico de geradores baseados em *templates* citado na Figura 2.

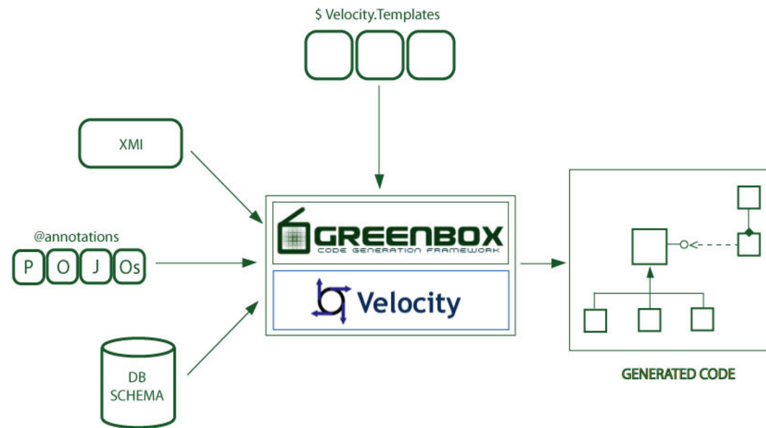


Figura 3 Modelo do gerador GreenBox
Fonte https://greenbox.dev.java.net/ (2007)

2.2.1 Arquivo XMI

O modelo UML pode ser transformado em arquivo XMI, que é um padrão OMG (*Object Management Group*) criado especificamente para a troca de informações entre diversas ferramentas. Assim, é possível utilizar a informação contida nesse arquivo para geração de código. Este arquivo XMI, é um arquivo XML (eXtensible Markup Language) que contém todos os dados que foram inseridos no modelo UML.Um arquivo XML é uma recomendação da W3C-World Wide Web Consortium (1999) para representar informações em uma estrutura específica, podendo descrever informações de documentos, livros, configurações de software, troca de um software para outro, banco de dados e

outros. A W3C especifica nomenclaturas para os componentes de um arquivo XML, listadas a baixo:

Elementos: conhecidas pelo nome de *tag,* primeiro nome depois do sinal de menor (“<”), terminada com sinal de maior (“>”). No arquivo XML existe início e fim de elemento. O início do elemento é especificado entre os sinais de maior e menor, exemplo: <elemento>, já o fim de elemento logo após o sinal de menor aparece uma barra (“/”), exemplo: </elemento>;

Valor de elementos:encontra-se entre o início e fim de um elemento, exemplo: <elemento>valor de elemento</elemento>

Atributos: são características de um elemento, aparece depois do nome de um início de elemento dentro dos sinais de menor e maior, exemplo: <elemento identificador = ‘1’>.

Valor de atributos: são, por definição, os valores que os atributos possuem. No exemplo do item 3 o valor do atributo *identificador* é ‘1’.

O gerador automático de código pode utilizar como base de dados diversos diagramas, dentre eles, o diagrama de classes e o diagrama de estados.

A seguir é apresentado (Figura 4) um diagrama de classe com classes, atributos e associações. A Figura 5 mostra como está organizada esta modelagem, seguindo uma estrutura de pacotes.

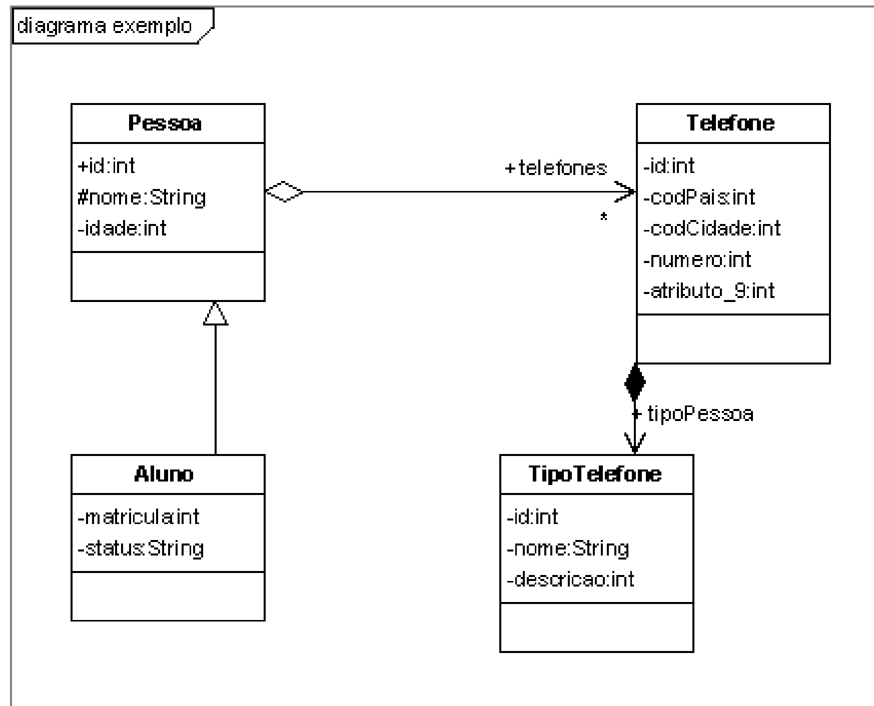


Figura 4 Exemplo de diagrama de classes

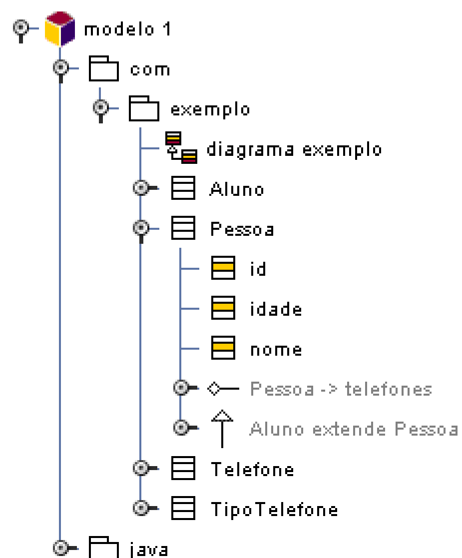


Figura 5 Estrutura da modelagem UML do exemplo

Mostra-se a seguir como as informações do modelo UML são descritas no Arquivo XMI:

A Figura 6 apresenta na linha 10 o elemento *UML:Model* com atributo *name* de valor 'modelo 1', nome do modelo, que é representado na Figura 4 o início da estrutura. A linha 13 (Figura 6) o elemento *UML:Package*, que possui o valor 'com' no atributo *name*, representando o pacote 'com' do modelo dentro do arquivo XMI. Este pacote possui um sub pacote nomeado de 'exemplo' (linha 16 da Figura 6).

```
10    <UML:Model xmi.id = 'Im7b156a91m1285e5bc2camm7e78' name = 'modelo 1' isSpecification = 'false'
11      isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
12      <UML:Namespace.ownedElement>
13        <UML:Package xmi.id = 'Im7b156a91m1285e5bc2camm7def' name = 'com' isSpecification = 'false'
14          isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
15          <UML:Namespace.ownedElement>
16            <UML:Package xmi.id = 'Im7b156a91m1285e5bc2camm7d2d' name = 'exemplo' visibility = 'public'
17              isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
```

Figura 6 Modelo e pacotes, informações do modelo UML no arquivo XMI

A Figura 7 apresenta o conteúdo do pacote ‘exemplo’. As classes TipoTelefone (linha 19), Pessoa (linha 44), Telefone (linha 84) e Aluno (linha 193).Os nomes das classes encontram-se no atributo *name* dentro dos elementos *UML:Class*. O pacote possui, também, as associações que são realizadas entre as classes (linhas 123 e 158).

```
16   <UML:Package xmi.id = 'Im7b156a91m1285e5bc2camm7d2d' name = 'exemplo' visibility = 'public'
17     isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
18     <UML:Namespace.ownedElement>
19       <UML:Class xmi.id = 'Im7b156a91m1285e5bc2camm7e4a' name = 'TipoTelefone'
20         visibility = 'public' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
21         isAbstract = 'false' isActive = 'false'>
44       <UML:Class xmi.id = 'Im7b156a91m1285e5bc2camm7e70' name = 'Pessoa' visibility = 'public'
45         isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
46         isActive = 'false'>
84       <UML:Class xmi.id = 'Im7b156a91m1285e5bc2camm7e5d' name = 'Telefone' visibility = 'public'
85         isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
86         isActive = 'false'>
123      <UML:Association xmi.id = 'Im7b156a91m1285e5bc2camm7e31' isSpecification = 'false'
124        isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
158      <UML:Association xmi.id = 'Im7b156a91m1285e5bc2camm7e0e' isSpecification = 'false'
159        isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
193      <UML:Class xmi.id = 'I194dd458m1285e778665mm7cc7' name = 'Aluno' visibility = 'public'
194        isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
195        isActive = 'false'>
209      <UML:Generalization xmi.id = 'I194dd458m1285e778665mm7cb4' isSpecification = 'false'>
```

Figura 7 Conteúdo do pacote ‘exemplo’ representado no arquivo XMI

A Figura 8 mostra o conteúdo da classe ‘Pessoa’. Observam-se os três atributos da classe, sendo eles ‘id’, ‘nome’ e ‘idade’. Estes são valores identificados pelo atributo *name* nos elementos *UML:Attribute*. Outras características apresentadas no modelo UML podem ser apontadas como, por exemplo, a visibilidade dos atributos por outras classes, esta característica é identificada pelo elemento *visibility*. Na Figura 4 os atributos ‘id’, ‘nome’, ‘idade’, são *public* (símbolo ‘+‘), *protected* (símbolo ‘#’) e *private* (símbolo ‘-‘) respectivamente, pode-se observar esses valores na Figura 8nas linhas 44, 50 e 56.

```
40  <UML:Class xmi.id = 'Im7b156a91m1285e5bc2camm7e70' name = 'Pessoa' visibility = 'public'
41    isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
42    isActive = 'false'>
43    <UML:Classifier.feature>
44      <UML:Attribute xmi.id = 'Im7b156a91m1285e5bc2camm7ded' name = 'id' visibility = 'public'
45        isSpecification = 'false' ownerScope = 'instance' changeability = 'changeable'>
50      <UML:Attribute xmi.id = 'Im7b156a91m1285e5bc2camm7dba' name = 'nome' visibility = 'protected'
51        isSpecification = 'false' ownerScope = 'instance' changeability = 'changeable'>
56      <UML:Attribute xmi.id = 'Im7b156a91m1285e5bc2camm7da8' name = 'idade' visibility = 'private'
57        isSpecification = 'false' ownerScope = 'instance' changeability = 'changeable'>
62    </UML:Classifier.feature>
63  </UML:Class>
```

Figura 8 Conteúdo da classe Pessoa: Atributos ‘id’, ‘nome’ e ‘idade’

Todos os elementos no arquivo XMI possuem um identificador (atributo *xmi.id),* isso permite que não ocorra repetição de informações, utilizando-se referência cruzada.Quando um elemento necessita de informação de outro elemento,ele pode realizar uma referência, inserindo como conteúdo, um elemento de nome igual ao do elemento referenciado, mas com apenas um atributo, que é o atributo *xmi.idref*. Este atributo deve possuir o mesmo valor que o atributo identificador (*xmi.id*) do elemento referenciado. A título de exemplo, a

Figura 9 mostra o elemento *UML:TypedElement.type*, que faz uma referência a um elemento *UML:DataType*, declarado em outro ponto do arquivo XMI.

Na classe Pessoa, o atributo ‘id’ necessita do valor do seu tipo ‘int’. No arquivo XMI essa informação de tipo do atributo encontra-se dentro do elemento *UML:TypedElement.type*(linhas 46 até 48 da Figura 9). Na linha 47 o elemento *UML:DataType* referência o elemento de mesmo nome (linha 183 da Figura 10) possuindo o valor mesmo do atributo *xmi.id* (linha 183 da Figura 10) em seu atributo *xmi.idref*(linha 47 da Figura 9).

Os tipos de atributos no arquivo XMI são classificados por primitivos e complexos. Os primitivos são identificados pelos elementos *UML:DataType* e os complexos pelos elementos *UML:Class,* que são atributos cujo tipo é uma outra classe.

Portanto, para descobrir o tipo do atributo, deve-se buscar primeiro o *xmi.idref* (destacado linha 47 na Figura 9) dentro do elemento *UML:TypedElement.type*, depois procurar o elemento dentro do arquivo XMI que possui no atributo *xmi.id*(destacado na linha 183 da Figura 10)o mesmo valor do *xmi.idref,* e recuperar o valor do atributo *name.*

```
44  <UML:Attribute xmi.id = 'Im7b156a91m1285e5bc2camm7ded' name = 'id' visibility = 'public'
45    isSpecification = 'false' ownerScope = 'instance' changeability = 'changeable'>
46    <UML:TypedElement.type>
47      <UML:DataType xmi.idref = 'Im7b156a91m1285e5bc2camm7df1'/>
48    </UML:TypedElement.type>
49  </UML:Attribute>
```

Figura 9 Identificação do tipo do atributo ‘id’ através de um identificador de referência

```
177  <UML:Package xmi.id = 'Im7b156a91m1285e5bc2camm7d2e' name = 'java' visibility = 'public'
178   isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
179   <UML:Namespace.ownedElement>
180     <UML:Package xmi.id = 'Im7b156a91m1285e5bc2camm7df0' name = 'lang' isSpecification = 'false'
181      isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
182      <UML:Namespace.ownedElement>
183        <UML:DataType xmi.id = 'Im7b156a91m1285e5bc2camm7df1' name = 'int' isSpecification = 'false'
184          isRoot = 'false' isLeaf = 'false' isAbstract = 'false'/>
185        <UML:DataType xmi.id = 'Im7b156a91m1285e5bc2camm7dee' name = 'void' isSpecification = 'false'
186          isRoot = 'false' isLeaf = 'false' isAbstract = 'false'/>
187        <UML:Class xmi.id = 'Im7b156a91m1285e5bc2camm7da9' name = 'String' isSpecification = 'false'
188          isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'/>
189      </UML:Namespace.ownedElement>
190    </UML:Package>
```

Figura 10 Destacado o identificador do atributo ‘int’

A Figura 11 apresenta as informações de uma associação, especificamente da associação da classe ‘Pessoa’ até a classe ‘Telefone’. O elemento *UML:Association* representa a associação em si. Ele possui dois sub elementos que representam os términos da associação nomeados de *UML:AssociationEnd* (linhas 106 e 121). Após a Figura 11 serão mostradas cada uma das terminações em detalhe.

```
103  <UML:Association xmi.id = 'Im7b156a91m1285e5bc2camm7e31' isSpecification = 'false'
104   isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
105   <UML:Association.connection>
106     <UML:AssociationEnd xmi.id = 'Im7b156a91m1285e5bc2camm7e37' visibility = 'public'
107      isSpecification = 'false' isNavigable = 'false' ordering = 'unordered' aggregation = 'aggregate
108      targetScope = 'instance' changeability = 'changeable'>
121     <UML:AssociationEnd xmi.id = 'Im7b156a91m1285e5bc2camm7e34' name = 'telefones'
122      visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering = 'unordered'
123      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
136   </UML:Association.connection>
137  </UML:Association>
```

Figura 11 Conteúdo de uma associação

A Figura 12 mostra o término da associação (Pessoa – Telefone) que se encontra na classe Pessoa representado no Arquivo XMI. Esse elemento UML:AssociationEnd possui característicasque podem ser listadas como as mais importantes aquelas apresentadas na Tabela1:

ATRIBUTO	CARACTERÍSTICA	VALORES
isNavegate:	Possui a 'seta' nesse término.	*true* ou *false*
aggregation	tipo da associação: agregação, composição ou sem identificação	*aggregate*, *composite* ou *none*
visibility	Visibilidade do fim da associação(utilizado para atributos) .	*public*, *private*ou *protected*

Tabela1Valores e características de um término da associação

Outras características podem aparecer dentro do elemento, uma delas é a cardinalidade do término da associação como, por exemplo, dentro do elemento *UML:MultiplicityRange* (linha 132, Figura 12), pelos atributo *lower* (mínima cardinalidade) e *upper* (máxima cardinalidade). Para identificar qual classe esse término da associação pertence, deve-se apontar através do identificador de referência (*xmi.idref)* abaixo do elemento *UML:AssociationEnd.participant* dentro de *UML:Class* (linha 138, Figura 12).

```
126 <UML:AssociationEnd xmi.id = 'Im7b156a91m1285e5bc2camm7e37' visibility = 'public'
127   isSpecification = 'false' isNavigable = 'false' ordering = 'unordered'
128   aggregation = 'aggregate' targetScope = 'instance' changeability = 'changeable'>
129   <UML:AssociationEnd.multiplicity>
130     <UML:Multiplicity xmi.id = 'Im7b156a91m1285e5bc2camm7e35'>
131       <UML:Multiplicity.range>
132         <UML:MultiplicityRange xmi.id = 'Im7b156a91m1285e5bc2camm7e36'
133         lower = '1' upper = '1'/>
134       </UML:Multiplicity.range>
135     </UML:Multiplicity>
136   </UML:AssociationEnd.multiplicity>
137   <UML:AssociationEnd.participant>
138     <UML:Class xmi.idref = 'Im7b156a91m1285e5bc2camm7e70'/>
139   </UML:AssociationEnd.participant>
140 </UML:AssociationEnd>
```

Figura 12 Conteúdo da ponta de inicio da associação (Pessoa – Telefone)

A Figura 13 mostra como o arquivo XMI descreve uma generalização. Essa generalização é exemplificada pela classe ‘Aluno’ que é uma especialização da classe ‘Pessoa’. Observa-se o conteúdo da classe ‘Aluno’ possui o elemento *UML:Generalization* com o identificador de referência (*xmi.idref,* linha 197*)* para a generalização (linha 209). Na generalização identifica-se a classe filha (*UML:Generalization.child*) possui o *xmi.idref* de elemento *UML:Class* para a classe ‘Aluno’, e a classe pai (*UML:Generalization.parent*) para classe ‘Pessoa’.

```
193  <UML:Class xmi.id = 'I194dd458m1285e778665mm7cc7' name = 'Aluno'
194  visibility = 'public' isSpecification = 'false' isRoot = 'false'
195  isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
196    <UML:GeneralizableElement.generalization>
197      <UML:Generalization xmi.idref = 'I194dd458m1285e778665mm7cb4'/>
198    </UML:GeneralizableElement.generalization>
199    <UML:Classifier.feature>
208  </UML:Class>
209  <UML:Generalization xmi.id = 'I194dd458m1285e778665mm7cb4'
210  isSpecification = 'false'>
211    <UML:Generalization.child>
212      <UML:Class xmi.idref = 'I194dd458m1285e778665mm7cc7'/>
213    </UML:Generalization.child>
214    <UML:Generalization.parent>
215      <UML:Class xmi.idref = 'Im7b156a91m1285e5bc2camm7e70'/>
216    </UML:Generalization.parent>
217  </UML:Generalization>
```

Figura 13 Generalização no arquivo XMI, classe Aluno herda de classe Pessoa

O mesmo feito até o momento para o diagrama de classes pode ser feito para os diversos diagramas UML, incluindo o diagrama de estados (Figura 14), como representado abaixo.

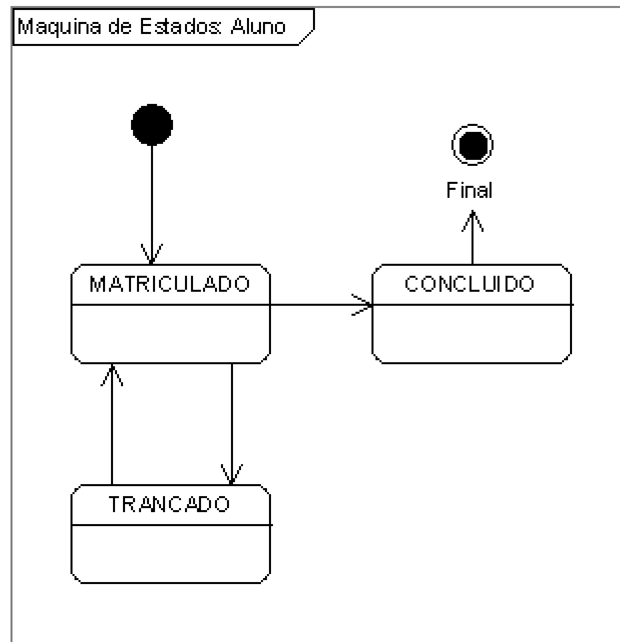


Figura 14 Máquina de estados de Aluno

A Figura 15 mostra como o diagrama de estados é descrito no arquivo XMI pelo elemento *UML:StateMachine* (linha 221). Os sub elemento *UML:Region.subvertex*possui os estados, o *UML:Region.transtion*, as transições da máquina de estados serão descritos nas Figura 16 e Figura 17.

```
221      <UML:StateMachine xmi.id = 'I194dd458m1285e778665mm7c88' name = 'AlunoStateMachine'
222        visibility = 'public' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
223        isAbstract = 'false' isActive = 'false'>
224        <UML:StateMachine.region>
225          <UML:Region xmi.id = 'I194dd458m1285e778665mm7c87' name = 'TopRegion_1'
226            visibility = 'public' isSpecification = 'false'>
227            <UML:Region.subvertex>
270            <UML:Region.transition>
317          </UML:Region>
318        </UML:StateMachine.region>
319      </UML:StateMachine>
```

Figura 15 Conteúdo da Maquina de Estado de Aluno.

A Figura 16 mostra os estados MATRICULADO, CONCLUIDO e TRANCADO, identificados pelo elemento *UML:State* nas linhas 228, 239 e 260 respectivamente, o estado inicial pelo elemento *UML:Pseudostate* (linha 254) com o atributo *kind* (linha 255)de valor '*initial'* e o estado final pelo elemento *UML:FinalState* (linha 248).

```
227  <UML:Region.subvertex>
228    <UML:State xmi.id = 'I194dd458m1285e778665mm7c7f' name = 'MATRICULADO'
229      visibility = 'public' isSpecification = 'false'>
239    <UML:State xmi.id = 'I194dd458m1285e778665mm7c74' name = 'CONCLUIDO' visibility = 'public'
240      isSpecification = 'false'>
248    <UML:FinalState xmi.id = 'I194dd458m1285e778665mm7c3f' name = 'Final' visibility = 'public'
249      isSpecification = 'false'>
254    <UML:Pseudostate xmi.id = 'I194dd458m1285e778665mm7c25' name = '' visibility = 'public'
255      isSpecification = 'false' kind = 'initial'>
260    <UML:State xmi.id = 'I194dd458m1285e778665mm7c0b' name = 'TRANCADO' visibility = 'public'
261      isSpecification = 'false'>
269  </UML:Region.subvertex>
```

Figura 16 Conteúdo do elemento *UML:Region.subvertex*, os estados do diagrama.

A Figura 17 mostra o conteúdo do estado 'MATRICULADO' e podem ser observadas as referências para as transições deste estado. As transições de saída encontram-se dentro do elemento *UML:Vertex.outgoing* representadas pela referência (*xmi.idref)* dos elementos *UML:Transition* (linha 231 e 232), e as transições de entrada no elemento *UML:Vertex.incoming* (linhas 235 e 237). O elemento *UML:PseudoState* possui apenas transições de saída. O elemento *UML:FinalState* possui apenas transições de entrada.

```
228  <UML:State xmi.id = 'I194dd458m1285e778665mm7c7f' name = 'MATRICULADO'
229    visibility = 'public' isSpecification = 'false'>
230    <UML:Vertex.outgoing>
231      <UML:Transition xmi.idref = 'I194dd458m1285e778665mm7c69'/>
232      <UML:Transition xmi.idref = 'I194dd458m1285e778665mm7beb'/>
233    </UML:Vertex.outgoing>
234    <UML:Vertex.incoming>
235      <UML:Transition xmi.idref = 'I194dd458m1285e778665mm7c00'/>
236      <UML:Transition xmi.idref = 'I194dd458m1285e778665mm7c20'/>
237    </UML:Vertex.incoming>
238  </UML:State>
```

Figura 17 Conteúdo do elemento UML: State com as referências das transições.

A Figura 18apresenta a transição pelo elemento *UML:Transition*. O sub elemento *UML:Transition.source* (linha 273) possui a referência (*xmi.idref*) para o elemento *UML:State* (linha274) de origem da transição, já o sub nó *UML:Transition.target* (linha 276).

```
271  <UML:Transition xmi.id = 'I194dd458m1285e778665mm7c69' visibility = 'public'
272    isSpecification = 'false' kind = 'external'>
273    <UML:Transition.source>
274      <UML:State xmi.idref = 'I194dd458m1285e778665mm7c7f'/>
275    </UML:Transition.source>
276    <UML:Transition.target>
277      <UML:State xmi.idref = 'I194dd458m1285e778665mm7c74'/>
278    </UML:Transition.target>
279  </UML:Transition>
```

Figura 18 Informações das transições no arquivo XMI

Portanto, o arquivo XMI apresenta todas as informações contidas nos diagramas UML. Sendo ele também um arquivo XML, pode-se fazer uso de ferramentas de manipulação de dados de arquivos XML para extrair essas informações como XSLT, JAXB (*Java Architecture for XML Binding*) desenvolvida por Mclaughin(2002) e outras. O maior impasse para extrair

informações dos arquivos XMI é que este utiliza informações cruzadas (mostrado na Figura 9), o que dificulta o entendimento de como os dados do diagrama estão apresentados no arquivo.

2.3 Iguassu Framework

O framework Iguassu (MVC) é uma implementação Java da arquitetura Iguassu. Até o presente momento, ainda não se tornou um produto comercial, já que ainda possui algumas limitações técnicas. O Iguassu está dividido com a camada de View no lado do cliente, como a maioria dos frameworks MVC utilizados na web, e não do servidor web (Figura 19).

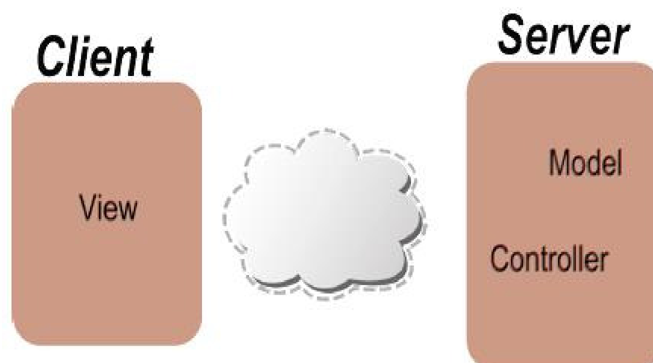


Figura 19 Modelo MVC utilizado em aplicação web

Segundo Canoza(2009), o lado servidor funciona como um provedor de serviços em um modelo SOA (*Service-Oriented Architecture*). Internamente o servidor é organizado em camadas, seguindo uma estrutura hierárquica baseada na idéia de que um sistema pode ter diversas aplicações, uma aplicação pode estar dividida em diversos módulos, e cada módulo da aplicação provê diversos serviços.

Camada de Aplicação:

A camada de Aplicação possui apenas uma classe por aplicação que é executada em um servidor, e que é a classe de controle principal. O Iguassu permite várias implementações de classes principais de controle, mas usa-se uma implementação baseada em outro framework MVC, livre, denominado *Struts*(APACHE, 2010). O *Struts* é configurado para que receba as solicitações HTTP(*Hyper Text Transfer Protocol*), pelo *HTTPRequest* e as mapeie todas para uma mesma ação. Esta ação é chamada de *HighlanderAction* no Iguassu framework. A *HighlanderAction* é apenas uma conexão do Iguassu com o Struts. Ela apenas transfere a requisição do Struts para o controlador frontal da aplicação. O controlador frontal é parte do Iguassu, ele implementa o padrão de projeto *FrontController*, definido pela especificação Entreprise Java Beans 3.0 (EJB 3.0). Todas as requisições passam pelo *FrontController*, e este cuida do controle de acesso, internacionalização, etc.

Camada de Módulos:

Um Módulo é um conjunto de serviços, e implementa o padrão *Business Delegate*, ou seja, seu papel é apenas transmitir a demanda de serviço diretamente para um Serviço de Aplicação.

Camada de Serviço:

Neste trabalho os serviços foram separados em dois grupos possíveis: Os serviços de acesso a dados, *Data Services*, e os serviços de aplicação, *Application Services*. Um *Data Service* é um serviço de manipulação de dados de uma única entidade do sistema. Um *Application Service*manipula várias entidades da Camada de Dados.Por conta da dificuldade de se representar os

*Application Services*em diagramas, é necessários que os mesmos sejam desenvolvidos manualmente.

Camada de Dados:

A Camada de Dados representa o domínio da aplicação. É esta camada que valida dados e os persiste em bancos de dados. Existem hoje frameworks apropriados para a persistência de objetos, e o framework Iguassu é baseado em tais frameworks. O Iguassu exige que, para cada classe do diagrama de classes UML, sejam implementadas doze classes Java, de forma a se utilizar padrões de projeto que permitam a padronização e conseqüente geração do código.

A figura a seguir mostra a arquitetura Iguassu:

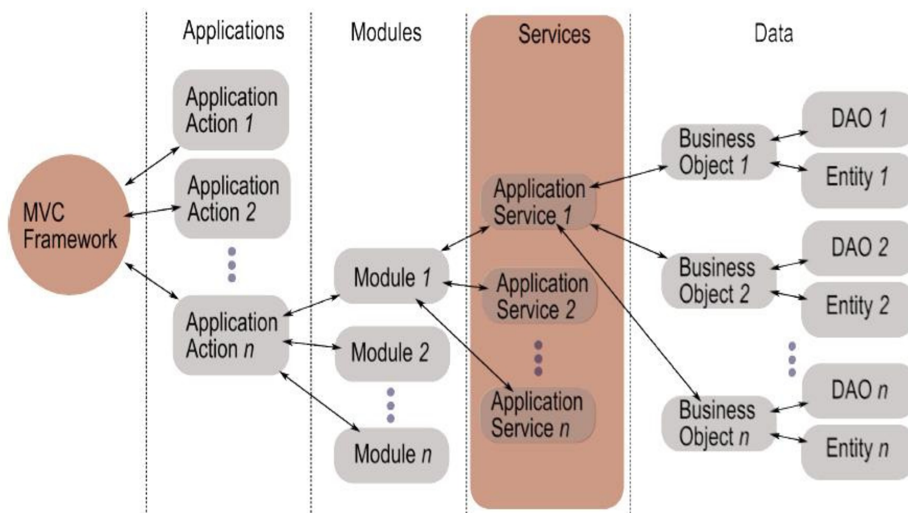


Figura 20 Arquitetura Iguassu, com destaque na única camada que necessita implementação manual

A única camada que necessita de implementação manual é a camada de serviços(*Application Services*).Os *Data Services*e todasas outras camadas podem

ser geradas através da construção de um gerador que utiliza informações contidas nos diagrama UML.

3 MATERIAL E MÉTODOS

Neste capitulo, são apresentados as características e procedimentos metodológicos deste trabalho. Na seção 3.1, classifica este trabalho descrevendo os aspectos relacionados à metodológica cientifica. Na seção 3.2 apresenta o método adotado para realização deste trabalho.

3.1 Tipo de Pesquisa

Quanto à natureza da pesquisa, trata-se de uma pesquisa aplicada na área de Tecnologia da Informação, por apresentar uma aplicação prática do trabalho proposto. Sua abordagem é qualitativa,pois usa contextos de uma situação natural como dados primários, tomando como ponto de partida um problema. É exploratória, por avaliar uma nova abordagem para este problema, havendo necessidade na descrição de todos os passos da pesquisa: o delineamento, a coleta de dados, a transcrição e sua preparação para análise dos resultados, caracterizando-a como descritiva-explicativa.

O trabalho foi realizado no Departamento de Ciência da Computação da Universidade Federal de Lavras no período de janeiro de 2009 a abril de 2010.

3.2 Método

O desenvolvimento do tema foi realizado a partir do modelo metodológico de Estudo de Caso. Está baseado na interpretação de dados primários coletados e obtidos diretamente na origem do evento pesquisado. Os dados secundários foram obtidos a partir de revisão da Literatura sobre o tema em artigos técnicos, relatórios, normas técnicas e referências sobre o assunto.

Inicialmente foi realizado um estudo a respeito da metodologia MDA e, mais especificamente, da modelagem UML, em portais eletrônicos, livros e artigos que discorressem sobre o tema. Verificou-se o impacto da modelagem UML na geração de arquivo de dados XMI e, então, criou-se um procedimento

de modelagem baseado na mesma.Em seguida, foi estudado o comportamento das classes do framework Iguassu, a fim de criar *templates* que se adequassem ao mesmo, após a implementação dos geradores de código automático.

Em relação à criação de geradores de código automático, a princípio foi estudada a tecnologia XSLT e, então, criado um gerador baseado na mesma. Ao aprofundar-se o conhecimento na área de geração de código, elencando-se, também, a tecnologia *Velocity*, foi implementado um novo gerador, denominado BlueBox, adotado devido a sua maior eficácia.Finalizada a criação do BlueBox e dos procedimentos de modelagem, os mesmos foram aplicados na empresa Mitah Technologies verificando-se a necessidade de modificações que foram implementadas durante o processo.

4 RESULTADOS

Neste capítulo são apresentados os resultados obtidos neste projeto,dividindo-o em três seções: CodeGen, BlueBox e Procedimentos de Modelagem.

Na primeira seção descreve-se o gerador automático de código utilizando tecnologia XSLT. Em seguida define-se a utilização da tecnologia *Velocity* para a construção do gerador BlueBox. Por fim são listados os métodos para modelagem do diagrama de classes e diagrama de estados.

4.1 CodeGen

O CodeGen (gerador)foi desenvolvido utilizando tecnologia Java. Este trabalha processando folhas de estilos, onde são definidas as regras que descrevem como serão apresentados os códigos no arquivo de saída (código gerado), associadas com o arquivo de base de dados, gerado pela ferramenta utilizada na construção do diagrama. O resultado do processamento será a geração do código de cada classe, distribuindo-as em seus pacotes e relacionando-as, quando necessário,de acordo com uma aplicação que utilize o modelo do Iguassu. Este gerador foi nomeado de CodeGen.

4.1.1 Arquitetura

Para a manipulação dos dados do arquivo XMI, citado anteriormente no referencial teórico, é utilizado o XSL (*EXtensible Stylesheet Language*) , uma linguagem de folha de estilo que descreve como o XML deve ser apresentado.

No XSL utilizamos o XSLT (*eXtensible Stylesheet Language Transformation*) e o XPath recomendado pela W3C (W3C, 1999): XSLTpara a transformação de informações de documentos XML em outros formatos (Java e XML) e XPath para navegar dentro de documentos que utilizem a linguagem XML, buscando nestes, elementos e atributos (dados).

Dentro do arquivo XMI, como foi mencionado, encontra-se toda a base de dados do modelo UML. A folha de estilo XSL é utilizada como *template* para a geração de código do gerador CodeGen. Por meio desta, é utilizado o XPath para buscar informações dentro deste arquivo XMI e, com a transformação do XSLT, estes dados são apresentados em um arquivo XML conforme definido no XSL.

É, então, criado um XSL com conteúdo específico conforme o padrão de cada tipo de classe do framework Iguassu. No gerador XSLT em questão, foi utilizada uma ferramenta XSLT para fazer a transformação do código, o SAXON (2010).

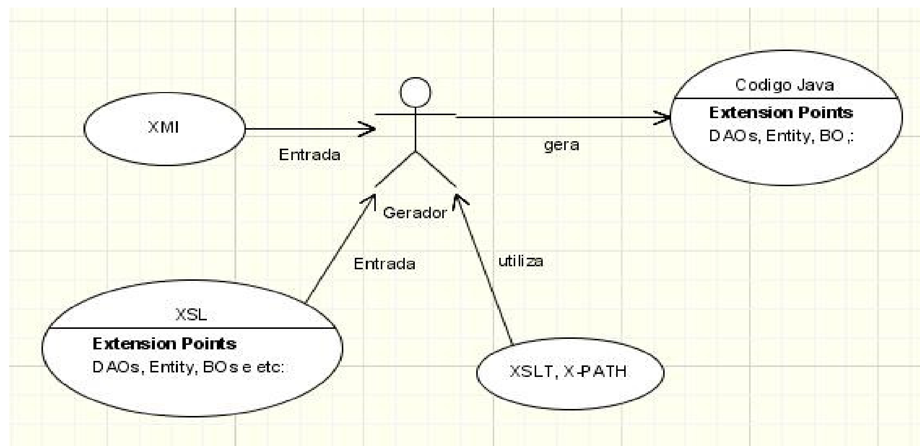


Figura 21 Funcionamento do gerador
Fonte CASTRO (2009)

O XSLT realiza a transformação criando um únicoarquivo XML com todo o código Java de saída devidamente organizado, devido esta característica foi criado uma estrutura XML para o resultado da transformação, para que este fosse processado e divido as informações em vários arquivos de saída. Logo após, o arquivo XML passa pelo File Processor que realizará análise (parse) do

arquivo separando o conteúdo do código gerado e gravando-o em arquivos em determinados locais, os quais são identificados durante o processo de análise do arquivo XML.

4.1.1.1 Processamento da geração de código

O arquivo XML gerado, possui uma estrutura bem definida, conforme exemplificado abaixo (Figura 22 Estrutura do arquivo XML gerado):

Figura 22 Estrutura do arquivo XML gerado

Após a transformação, o arquivo XML possui a estrutura acima e no arquivo XSL são definidos os valores de cada nó desta estrutura, cujos significados são:

<files>:É um único elemento no arquivo XML, sendo o que representa o conteúdo da transformação realizada na folha de estilo com o arquivo XMI.

<subPackage>: Este elemento representa os subpacotes dos módulos (do framework Iguassu) onde será gravado o código gerado.

<file>: Para cada elemento é gerado um arquivo. Este arquivo possuirá o código gerado**.**

<name>: O valor presente neste elemento refere-se ao nome do arquivo gerado.

<module>: Este elemento representa o módulo em que o arquivo gerado será gravado.

<content>: O valor presente neste elemento será o conteúdo do arquivo gerado.

O arquivo resultante é processado pela ferramenta JAXB, que permite extrair dados de arquivos XML, criando arquivos com os conteúdos de cada elemento *file*, gravando-os dentro de seus módulos correspondentes (valor do elemento *module).* Caso necessário é utilizado o elemento *subPackage* para criar sub pacotes dentro dos módulos. Assim é possível realizar várias gerações com um único *template* XSL.

4.1.1.2 Folhas de estilo (XSL)

Serão apresentado alguns exemplos simples da implementação das folhas de estilos.

A Figura 23 descreve como será a declaração de uma classe no arquivo gerado. O conteúdo elemento *xsl:template* (linha 1 na Figura 23) é aplicado toda vez que encontra um elemento *UML:Class* (valor especificado no atributo *match* no elemento *xsl:template,* linha 1 da Figura 1)no arquivo XMI. A linha 3 da Figura 23verifica se a classe é uma classe abstrata com o elemento *xsl:if*, caso seja é escrito no arquivo de saída o valor *abstract*(linha 4). A linha 6 escreve o nome da classe pelo comando *xsl:value-of* com o atributo *select* de valor '*@name'*. A linha 8 até a 10 escreve o construtor da classe.

```
1  <xsl:template match="UML:Class" >
2      <!-- declaracao da classe -->
3      public  <xsl:if test="@isAbstract = 'true'">
4                  abstract
5              </xsl:if>
6                  class <xsl:value-of select="@name"/> {
7          <!-- conteudo da classe -->
8          public <xsl:value-of select="@name"/>() {
9
10         }
11         ...
12     }
13 </xsl:template>
```

Figura 23 Exemplo XSL para declaração de uma classe Java

Na Figura 23 não verifica se a classe é uma especialização de outra classe, pois necessita utilizar referência cruzada (*xmi.idref*). Devido à dificuldade em obter dados por referência cruzada pelos *templates* XSL do arquivo XMI, nas Figuras Figura 24 até 26 descreve-se como é realizado a declaração de atributos em uma classe, que necessita obter o tipo do atributo, um exemplo de referência cruzada.

Na linha 8 da Figura 24 mostra como é chamado a aplicação do *template* (Figura 25) para cada atributo.

```
1  <xsl:template match="UML:Class">
2      <!-- declaracao da classe -->
3      ...
4      {
5      <!-- conteudo -->
6      ...
7      <!-- declara atributos -->
8      <xsl:apply-templates select="UML:Classifier.feature/UML:Attribute" />
9      }
10 </xsl:template>
```

Figura 24 Chamada do *template UML:Attribute* para declarar os atributos

A Figura 25 descreve o *template* chamado para cada atributo da classe. A linha 12 cria uma variável 'IDTipoAtributo' para guardar o identificador de referência do tipo do atributo (visto nas Figura 9 eFigura 10) que é recebido pelo comando *xsl:value-of* na linha 13. Na Linha 17, durante a aplicação do *template UML:DataType*(Figura 4), é selecionado apenas o elemento *UML:DataType* do arquivo XMI que possui o *xmi.id* igual a variável 'IDTipoAtributo'. A linha 18 passa por valor do nome do atributo e a linha 19 a visibilidade do atributo,como parâmetros para o *template UML:DataType.*

```
10 <xsl:template match="UML:Attribute" >
11     <!-- identificado do tipo do atributo para ser buscado-->
12     <xsl:variable name="IDdoTipoAtributo">
13         <xsl:value-of select="UML:TypedElement.type//@xmi.idref"/>
14     </xsl:variable>
15     <!-- aplica a declaracao dos atributos que possuem tipo de atributo
16     primitivo-->
17     <xsl:apply-templates select="//UML:DataType[@xmi.id = $IDdoTipoAtributo ]" >
18         <xsl:with-param name="nomeDoAtributo" select="@name"/>
19         <xsl:with-param name="visibilidade" select="@visibility"/>
20     </xsl:apply-templates>
21 </xsl:template>
```

Figura 25 *Template* UML:Attribute para declaração de atributos

A Figura 26 apresenta o *template* que é aplicado aos elementos *UML:DataType.* As linhas 25 e 26 são comandos para receber os valores do

nome e visibilidade passado com parâmetros. A linha 28 declara a variável com o nome do tipo do atributo. Em seguida são escritos no arquivo os valores da visibilidade, tipo do atributo e nome do atributo.

```
24 <xsl:template match="UML:DataType" >
25     <xsl:param name="nomeDoAtributo"/>
26     <xsl:param name="visibilidade"/>
27     <!-- variavel com o nome do tipo do atributo -->
28     <xsl:variable name='tipoDoAtributo'>
29         <xsl:value-of select='@name'/>
30     </xsl:variable>
31     <!-- Declaracao: visibilidade tipoDoAtributo nomeDoAtributo; -->
32     <xsl:value-of select="$visibilidade"/>
33         <xsl:value-of select='$tipoDoAtributo'/>
34             <xsl:value-of select="$nomeDoAtributo"/>;
35 </xsl:template>
```

Figura 26 *Template* no escopo do tipo de atributo que imprime a declaração do atributo

A Figura 27 apresenta a saída do código gerado para a classe Pessoa (apresentado no modelo exemplo na Figura 4) do *template* XSL com os comandos apresentados das Figura 23 até Figura 26.

```
1  public class Pessoa {
2
3      public Pessoa() {
4
5      }
6      private int id;
7      protected String nome;
8      public int idade;
9
10 }
```

Figura 27 Código gerado para a classe Pessoa, definida no *template* XSL

Portanto, com as folhas de estilo XSL é possível definir como será apresentado o código no arquivo de saída.

4.2 BlueBox

O BlueBox é uma ferramenta para geração automática de código. Ele utiliza-se do arquivo XMI, da mesma forma que o CodeGen, como entrada de dados para obter informações do modelo UML. Através da *engine Velocity*, o gerador processa os dados conforme as regras contidas nos *templates Velocity*, ocorrendo assim a geração do código. O tutorial de como utilizar BlueBox encontra-se no **Anexo 1** deste projeto.

4.2.1 Arquitetura

O BlueBox apresenta três componentes que são introduzidos abaixo, com maior aprofundamento de suas definições e funções posteriormente:

XMI Parse: Este componente tem como principal função obter os dados do arquivo XMI.

Classificador de *Templates*:por definição, classifica os *templates* conforme seu comportamento durante a geração.

Engine *Velocity*: Realiza o referenciamento das informações obtidas pelo arquivo XMI nos *templates Velocity* para a geração do código.

Assim, conforme especificado na figura abaixo (Figura 28), o BlueBox recebe como entrada um arquivo XMI e os *templates*. O arquivo XMI é, então, analisado pelo XMI parse, e este envia as informações para o engine *Velocity*. Ao mesmo tempo, os *templates* são classificados no Classificador de *Template* e, então enviados, também, para o engine *Velocity*. Este processa as informações conforme os *templates* gerando o código de saída (código gerado).

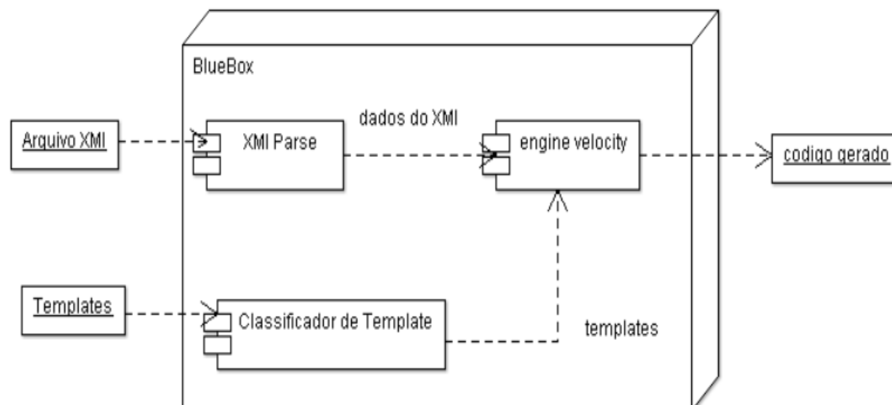


Figura 28 Arquitetura do BlueBox

A principal diferença entre o gerador CodeGen e o BlueBox é que o último realiza um pré-processamento das informações antes de enviá-las para os *templates*. Com isso os *templates* do BlueBox tornam-se mais legíveis que as folhas de estilos (*templates* XSL) facilitando a aprendizagem da programação dos *templates* e manutenção. Essa vantagem será observada no tópico *Templates Velocity*da Seção 4.2.2.1.

4.2.1.1 XMI Parse

O XMI Parse tem como objetivo buscar as informações no interior do arquivo XMI e transformá-las em instâncias de objetos. Este recebe o arquivo XMI e tem suas informações obtidas por meio da ferramenta JColtrane, utilizada para extrair dados de arquivos XML. Após isto, tais informações são organizadas no Organizer (Figura 29) e, então, enviadas para o Engine *Velocity* (conforme Figura 28apresentada).

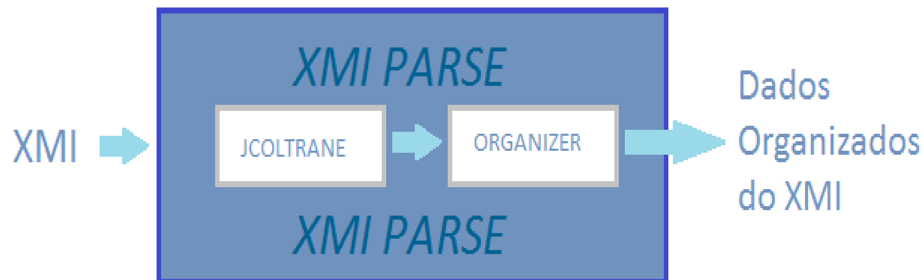


Figura 29 Componentes do XMI Parse

JColtrane

O JColtrane é uma ferramenta brasileira utilizada para analisar arquivos XML, desenvolvida por Nuccielli, 2008. Com ela foi possível obter as informações do arquivo XMI de forma simplificada, permitindo instanciar e manipular objetos Java com os dados do diagrama UML.

A Figura 30 mostra exemplo de como obter os dados do arquivo XMI e construir um objeto para uma classe *XMIClass*.Esta classe caracteriza todos elementos *UML:Class* do arquivo XMI. A linha 13 com a *annotation @StartElement* indica que o método startElementClass (linha 14) será executado toda vez que o JColtrane iniciar a leitura do elemento *UML:Class* (definido pelo valor de *tag* na linha 13) dentro do arquivo XMI. NA linha 16 o objeto *attHolder* recebe todos os valores dos atributos do elemento *UML:Class.* A linha 19 instância um objeto *XMIClass,* e logo em seguida este objeto recebe em seus atributos os valores dos atributos do elemento *UML:Class* pelo objeto *attHolder* (linha 20 até 23).

```java
13   @StartElement(tag = "UML:Class")
14   public void startElementClass(ContextVariables contextVariables) {
15
16       AttributesHolder attHolder = contextVariables.getLastEvent()
17               .getAtributesHolder();
18
19       XMIClass xmiClass = new XMIClass();
20       xmiClass.setId(attHolder.getValue("xmi.id"));
21       xmiClass.setName(attHolder.getValue("name"));
22       ...
23       xmiClass.setVisibility(attHolder.getValue("visibility"));
24   }
```

Figura 30 de como obter dados de um elemento UML:Class pelo JColtrane

Assim foram criadas as classes que representam os componentes do arquivo XMI necessários para a geração e classes com métodos que utilizam o JColtrane, possibilitando construir objetos conforme a estrutura apresentada pelos dados neste arquivo. Desta forma, a manipulação destes objetos torna-se facilitada durante a execução da ferramenta BlueBox.

Organizador de informação

Após a obtenção dos dados XMI pelo BlueBox por meio do JColtrane descrita anteriormente, estes dados passam por um Organizador de Informação.

A organização das informações é realizada a fim de facilitar a programação dos *templates Velocity*, pois o entendimento da estrutura dos dados nos arquivos XMI pode ser simplificado, já que realiza um algoritmo inicial (pré-algoritmo) não sendo necessário que o mesmo seja repetido diversas vezes durante a implementação do *template*. Assim, diminui-se o tempo de aprendizagem e acelera-se o de produção.

4.2.1.2 Gerador

O núcleo do gerador, que gerencia e manipula as informações, recebe os dados que foram obtidos pelo JColtrane e analisados pelo Organizador de Informação. Estas informações são, então, gerenciadas pelo Processador *Velocity*, que as envia para o Processador de Categorias, o qual realiza um comportamento específico durante a geração do código de acordo com a categoria do *Template Velocity*.

Os componentes do núcleo do gerador:Processador *Velocity* e Processador de Categorias; e os *Templates Velocity* serão descritos a seguir:

Processador *Velocity*

O Processador *Velocity*é o componente principal do gerador BlueBox,que funciona como um gerenciador de duas funções desta ferramenta: aplicar filtro nos dados e encaminhar os *templates* e os dados para o Processador de Categorias.

Antes de enviar os dados para o Processador de Categorias, o Processador *Velocity* realiza filtragens nos dados sendo elas:

- Filtragem de Classes: o Processador permite que o usuário escolha as classes do modelo UML que serão geradas;
- Filtragem de Pacotes: o Processador permite selecionar os pacotes que serão gerados do modelo;
- Filtragem de Categoria: seleciona quais são as categorias que serão geradas. Uma categoria possui um conjunto de *templates*, assim, apenas os *templates* daquela categoria serão gerados;
- Filtragem de *Templates*: seleciona os *templates* que serão gerados.

O Processador *Velocity* seleciona o processamento específico do Processador de Categorias e envia os *templates* e os dados para este componente.

Processador de Categorias

O processador de categorias realiza o processamento específico para cada *template* de acordo com sua categoria. As categorias definem o comportamento de como será gerado o código. Este comportamento está relacionado com as entradas de dados (tipos e quantidades) que os *templates* receberam durante a geração. Estas entradas de dados são denominadas contextos.

Os contextos são objetos que representam os dados do modelo e podem ser utilizados pelos *templates*. São contextos aplicados nos *templates*:

- ✓ *packages*: são os pacotes definidos no modelo da aplicação.
- ✓ *application:* possui todos os dados obtidos a partir da raiz do modelo UML.
- ✓ *package:* representa um pacote no modelo.
- ✓ *subPackage:* possui o caminho dos sub pacotes do módulo. Os nomes das pastas são concatenados por pontos (“.”).
- ✓ *class:* possui todos os dados da classe.
- ✓ *template*: possui os dados do *template* que está sendo gerado.
- ✓ *stateMachine*: contém todos os dados de um determinadodiagrama de estados.

As categorias e seus respectivos contextos podem ser divididas em:

Application: nesta categoria é gerado um único arquivo para toda a aplicação, e este *template* utiliza dados provenientes dos seguintes contextos:

- ✓ *packages*;

- ✓ *application*;
- ✓ *subPackage*;
- ✓ *template*.

Classes: é gerado um arquivo para cada classe do modelo.

Contextos:

- ✓ *class*;
- ✓ *statemachine:* máquina de estados da classe que está sendo gerada;
- ✓ *subPackage*.

Packages: é gerado um arquivo para cada pacote no modelo.

Contextos:

- ✓ *package;*
- ✓ *application;*
- ✓ *subPackage.*

SubPackages: é gerado um arquivo para cada sub pacote do modelo (a partir do pacotes)

Contextos:

- ✓ *subPackage;*
- ✓ *package.*

StateMachines: é gerado um arquivo para cada diagrama de estados do modelo.

Contextos:

- ✓ *stateMachine;*
- ✓ *subPackage;*

Processador de Variáveis de Caminho

Assim como o processador de categorias, existe o Processador de Variáveis de Caminho. As variáveis de caminho definem o caminho em que serão gravados os arquivos gerados. São utilizadas após os nomes das categorias criando-se uma pasta com o nome da variável, como explicado no tutorial (Anexo 1), com exemplos abaixo de algumas variáveis já criadas:

- ✓ _MODULEROOT_: o código será gerado no caminho raiz de determinado módulo (apps/nomeProjeto/nomeDoModulo/src/com/nomeDoProjeto nomeDoModulo/);

- ✓ _MAINROOT_: o código será gerado no caminho raiz do modulo main do projeto (*apps/nomeProjeto/main/src/com/ nomeDoProjeto/main*);

- ✓ _CONTEXT_: o código será gerado dentro da pasta 'context' do projeto *(/WEB-INF/src/contexts/nomeDoProjeto/*);

Templates Velocity

Os *templates*, como já foram citados, tem como função descrever como os arquivos gerados devem ser apresentados. Em suma, o comportamento dos *templates* durante a geração de arquivos é definido de acordo com sua categoria, função desempenhada pelo processador de categorias. As variáveis de caminhos, por sua vez, definirão o local onde o arquivo será gerado.

Abaixo pode-se visualizar alguns exemplos da utilização dos *template Velocity.*A Figura 31mostra com é realizada a declaração do pacote dentro da classe. Utiliza-se o contexto *class* e *subPackage.* No contexto *class,* recebe o valor de seu atributo *namePackage* (definido durante a organização da

informação) para definir o pacote do módulo e o valor do contexto *subPackage* para definir o sub pacote deste módulo.

```
1   package  ${class.namePackage}.${subPackage};
```

Figura 31Declaração de pacote

A Figura 32 mostra a utilização do atributo *interfaces* dentro de um laço (foreach) junto a um comando de condição. O laço realiza as operações para cada interface que a classe possui. A operação definida é uma condição que verifica se a interface é do tipo *ActivityInstance,* caso seja, esta interface será importada dentro da classe.

```
29  #foreach ($interface in $class.interfaces)
30      #if ($interface.name == 'ActivityInstance')
31      import com.mitahtech.iguassu.model.entity.ActivityInstance;
32      #end
33  #end
```

Figura 32 Iteração para cada inteface e verificando se existe a interface ActivityInstance.

Na Figura 33 o comando condicional verifica se a classe possui super classe através do atributo *hasSuperClass* do contexto *class.* Se este atributo possui valor verdadeiro, será importada a super classe.Se não, serão importadas as entidades padrão do framework Iguassu.

```
40  #if (${class.hasSuperClass})
41      import ${class.superClass.namePackage}.model.entity.${class.superClass.name};
42  #else
43      import com.mitahtech.iguassu.model.entity.GenericEntity;
44      import java.io.Serializable;
45  #end
```

Figura 33 Verificando se a classe possui super classe

A Figura 34 apresenta a maneira como os atributos primitivos de uma classe são listados. São denominados atributos primitivos aqueles que se encontram dentro do escopo da classe no diagrama de classes (atributos não associados).

```
115     #foreach (${attribute} in ${class.primitiveAttributes} )
116         ${attribute.visibility} ${attribute.dataType} ${attribute.name};
117     #end
```

Figura 34 Listando atributos primitivos

Na Figura 35 é exposta a maneira como são listados os atributos associados de uma classe. São classificados como atributos associados de uma classe todos aqueles que possuem o tipo de uma entidade que se encontra no diagrama de classes e está associada com a mesma.

```
115     #foreach (${attribute} in ${class.associateAttributes} )
116         ${attribute.visibility} ${attribute.dataType} ${attribute.name};
117     #end
```

Figura 35 Listando atributos associados

Na Figura 36 é apresentado como se lista todos os atributos de uma classe. O atributo *attributes* possui todos os atributos da classe.Nele estão listados os atributos primitivos e os associados.

```
119     #foreach (${attribute} in ${class.attributes} )
120         ${attribute.visibility} ${attribute.dataType} ${attribute.name};
121     #end
```

Figura 36Listando todos os atributos

A Figura 37 apresenta como listar todos os marcadores de uma classe e seus valores. Utiliza-se um laço recebendo todos os marcadores do atributo

tagsValue do contexto *class*. Neste atributo recebem-se os valores dos atributos *name* (nome do marcador) e *value* (valor do marcador).

```
1  #foreach ($tagValue in $class.tagsValue )
2      ${tagValue.name}     ${tagValue.value}
3  #end
```

Figura 37 Listando marcadores de uma classe

A Figura 38 apresenta como listar todos os marcadores e valores dos atributos associados de uma classe.

```
1  #foreach (${attribute} in ${class.associateAttributes} )
2      #foreach ($tagValue in $attribute.tagsValue )
3          ${tagValue.name}    ${tagValue.value}
4      #end
5  #end
```

Figura 38 Listando marcadores dos atributos associados de uma classe

4.3 Procedimentos de modelagem

O processo de modelagem necessita que o engenheiro de software utilize procedimentos para a construção do diagrama permitindo que o CodeGen e o BlueBox gerem os códigos de forma eficaz.

Neste trabalho, foram estabelecidos métodos de modelagem dos diagramas de classes e estados, especificamente, já que o framework Iguassu utiliza-se destes diagramas para a geração de código. Estes procedimentos serão descritos a seguir.

4.3.1 Diagrama de Classes

O diagrama de classes deve representar de forma sucinta e bem estruturada todos os dados necessários para o domínio de uma aplicação, conforme é feito em um modelo Entidade Relacionamento (ER), abstraindo todos os elementos essenciais para a geração de código como entidades (representadas pelas classes), atributos, associações, generalizações e outros, utilizando-se da sintaxe UML.

4.3.1.1 Classes

O arquiteto de software não deve modelar os comportamentos no diagrama. Todas as classes modeladas possuem serviços básicos, chamados *dataservices,* que podem ser gerados devido às padronizações adotadas pelo framework Iguassu. Assim, cada classe do diagrama possui os serviços de CRUD (*Create, Read, Update* e *Delete*).Tendo isto em vista, não é necessário modelar estes comportamentos. Outros serviços, que precisam ser implementados devido à complexidade de um determinado comportamento, são modelados em outros diagramas da UML.

As classes devem possuir seus atributos modelados na própria classe caso sejam atributos do tipo primitivos (como *int, double, float* e outros) ou do

tipo de outra classe.Porém, esta classe não deve ter sido modelada para a determinada aplicação (*String, Date* e outros). A Figura 39mostra a modelagem da classe "Pessoa", com seus atributos e seus tipos representados dentro da classe: o atributo 'idade' do tipo primitivo *int* e o 'nome' do tipo de outra classe *String*.

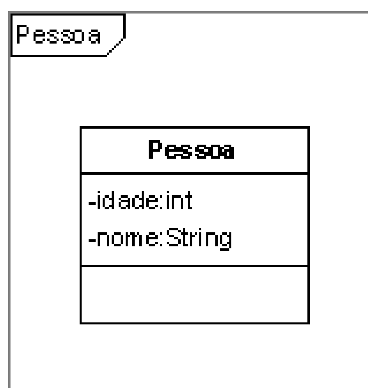


Figura 39 Exemplo de modelagem dos atributos dentro de uma classe

Os atributos do tipo de "outra classe" (possui associação) que estão no domínio da aplicação e não devem estar representados dentro da própria classe.Estes são explicados na Subseção 4.3.1.2.

Entidades *statemachine*

O framework Iguassu pode caracterizar uma entidade como máquina de estados (*statemachine*) quando esta possui uma realização com a interface *StateMachineEntity* do pacote Iguassu.A uma entidade que seja *statemachine,* deve-se adicionar o atributo *status* que indica qual o estado atual da entidade. Os estadossão definidos nos diagramas de estados que são descritos na Subseção 4.3.2. A Figura 40 mostra a realização da classe Aluno com a interface *StateMachineEntity* que define essa classe como *statemachine*.

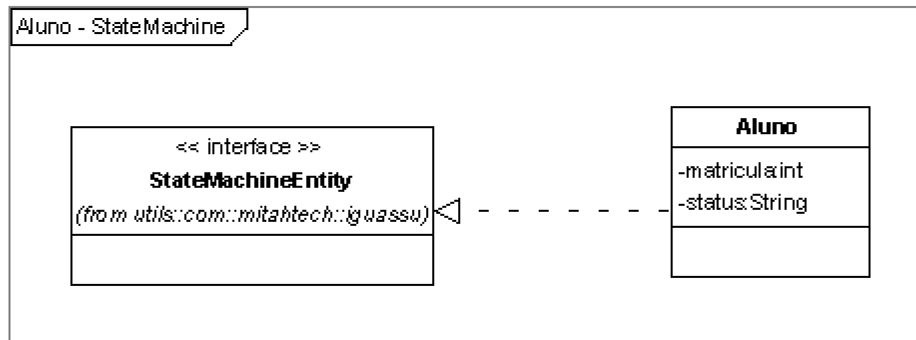


Figura 40 Entidade Aluno possui realização com StateMachineEntity

4.3.1.2 Associações

Atributos que sejam do tipo "outras classes"devem ser representados pelas associações nas quais são ligadas: a classe que possui o atributo com a classe do tipo do atributo. A associação possui dois términos: início e fim. O início identifica a classe que possui o atributo e deve ser indicado pelo tipo da associação, agregação ou composição, definida na UML. O fim é indicado onde a 'seta' da associação se encontra, identificando a classe do tipo do atributo.

A nomenclatura desses atributos é obtida através do nome da classe do tipo do atributo com a primeira letra minúscula.Caso seja necessário personalizar o nome do atributo, deve-se nomear o fim da associação com o nome desejado.

A Figura 41exemplifica a utilização da associação para criar um atributo do tipo Telefone dentro da classe Pessoa. É feita uma associação de agregação que inicia em Pessoa (classe que possui o atributo) e termina na classe Telefone (classe do tipo do atributo). Neste caso, foi personalizado o nome do atributo da classe Pessoa para 'telefones' nomeado no fim da associação, pois este atributo possui multiplicidade infinita (símbolo '*').Assim, durante a geração, o nome do atributo na classe Pessoa estará no plural (nome indicado para atributos de

multiplicidade infinita)ao invés de apenas ‘telefone’ (nome da classe do tipo do atributo com a primeira letra minúscula).

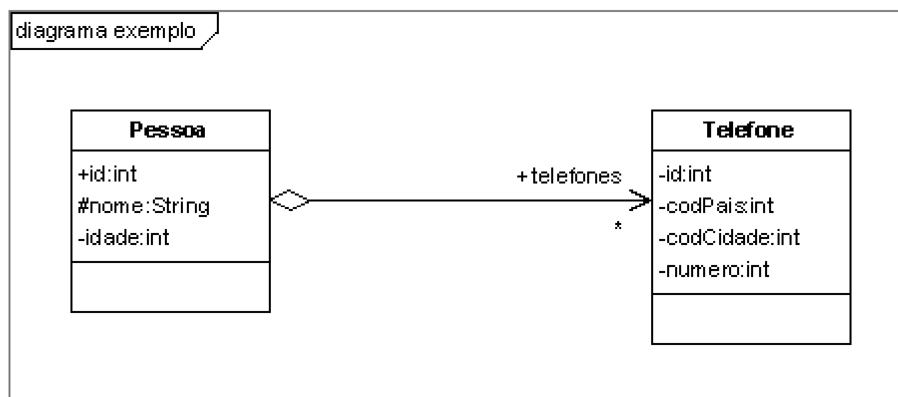


Figura 41 Representação do atributo ‘telefones’ da classe Pessoa em uma associação

No diagrama uma classe deve ter apenas uma associação de início na mesma com fim de associação em outra classe. Para criar mais de um atributo que seja do mesmo tipo, cria-se uma nova classe no diagrama, que representará características específicas do tipo daquele atributo e associa-se essa nova classe com a do tipo do atributo.

Nos procedimentos, não é permitida a criação de duas associações entre duas classes.Se fosse necessário, no exemplo da Figura 41, a classe Pessoa possuir dois atributos, ‘telefoneResidencial’ e ‘telefoneCelular’ do mesmo tipo (Telefone),devido àquela restrição, deve-se criar uma nova classe que terá características específicas da classe Telefone, chamada de TipoTelefone como mostra a Figura 42. Essa nova classe terá as características já cadastradas para TipoTelefone, podendo ter o atributo ‘nome’ com valor ‘Telefone Residencial’ e ‘Telefone Celular’. Assim, quando a classe Pessoa necessitar de um telefone residencial, o seu atributo ‘telefones’ deve possuir um objeto com o atributo ‘tipoTelefone’ específico para telefone residencial.

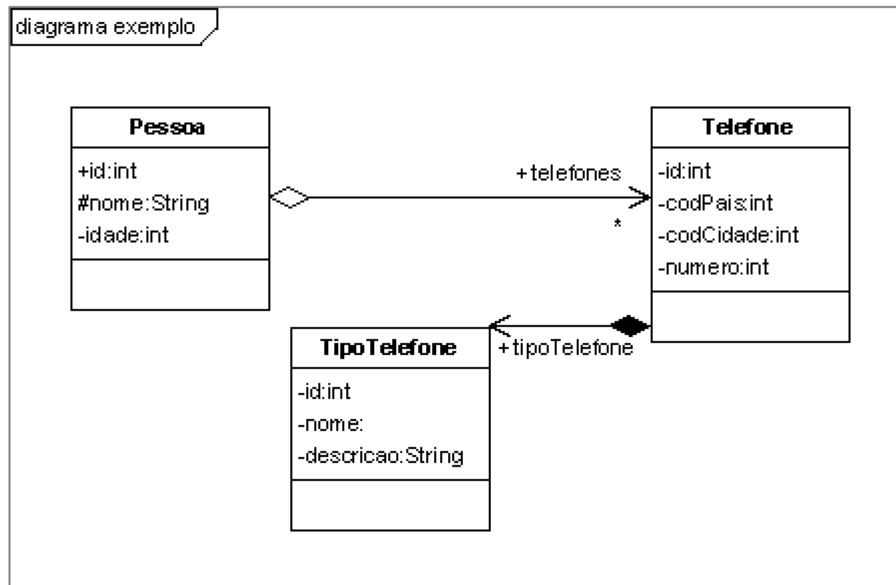


Figura 42Criação da classe TipoTelefone, não devendo criar mais de uma associação entre classes

4.3.1.3 Atributos

O framework Iguassu permite a utilização de validadores nos atributos de uma classe. Os validadores tem como objetivo verificar se os atributos estão de acordo com uma especificação do framework Iguassu descrita na documentação do Iguassu, e estes validadores são adicionados no atributo dentro do diagrama UML.

Para adicionar um validador em um atributo deve-se adicionar um marcador UML de nome *validator* e o valor deste marcador deve ser o nome do validador. A Figura 43 Utilização de validadores no atributo mostra a utilização do validador *NotNullValue* em um atributo.

Propriedades | Estilo | Documentação | SourceCode | Imposições | Valores marcados | C++ Properties

	Marcar	Valor
	validator	NotNullValue

Figura 43 Utilização de validadores no atributo

O Iguassu especifica que o validador *NotNullValue* antes de persistir os dados daquele atributo verifica se o atributo possui valor *Null* (nulo), caso o valor do atributo seja nulo, a entidade que possui esse atributo não será persistida.

Existem diversos validadores no Iguassu, inclusive validadores com parâmetros, mas isto é funcionalidade do framework, e não dos geradores. Assim, o Iguassu permite adicionar facilmente validadores nos atributos através do diagrama de classes.

4.3.1.4 Pacotes

O diagrama de classes deve ser modelado organizando as classes em pacotes conforme forem sendo implementadas. A utilização de pacotes permite, durante a geração de código, saber qual pacote uma determinada classe pertence. Assim, a classe gerada será gravada no caminho correto e terá em seu código a nomenclatura do pacote de acordo com o que foi modelado.

O primeiro pacote a ser criado identifica a aplicação que será gerada o código. Este pacote não é utilizado durante a geração no nome do pacote e caminho da classe gerada. A geração utiliza apenas os dados que estão depois do primeiro pacote. As classes que serão utilizadas como utilitários, que não possui característica de entidade, devem ser colocadas dentro do primeiro pacote de nome *utils*, os dados dentro deste pacote não serão gerados, porém podem ser

utilizados durante a geração de outras entidades, por exemplo na identificação de tipos de atributos.

A Figura 44 mostra a aplicação da estrutura de pacotes seguindo o exemplo do modelo da Figura 4 e Figura 5. Os primeiros pacotes *aplicacao1* e *aplicacao2* apresentam a existência de duas aplicações para geração dos códigos, isso permite identificar durante a geração qual aplicação deve-se gerar o código. A figura mostra também a utilização do pacote *utils*, colocando as classes do pacote *java* pois as classes do modelo exemplo(Figura 4) necessitam de classes deste pacote para a identificação dos tipos de atributos que são *String* e *int.*

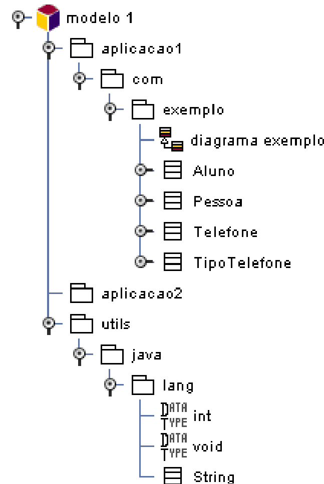


Figura 44 Estrutura de pacotes de acordo com os procedimentos

Portanto, a utilização desta estrutura de pacotes é essencial para a geração de códigos, permitindo identificar a nomenclatura dos pacotes nas classes e o local onde esta será gravada durante a geração.

4.3.2 Diagrama de Estados

Uma entidade *statemachine* necessita de um diagrama de estados para que o framework Iguassu identifique os estados e transições de estados que esta

entidade possui. Para associar um diagrama de estados com uma entidade *statemachine* o diagrama deve possuir o mesmo nome da entidade concatenado com ‘*StateMachine’*, permitindo que o gerador identifica que se a entidade ‘x’ possui a máquina de estados ‘xStateMachine’. A Figura 45 mostra a criação da máquina de estados AlunoStateMachine para a entidade Aluno, facilitando a identificação da máquina de estados da entidade Aluno durante a geração de código.

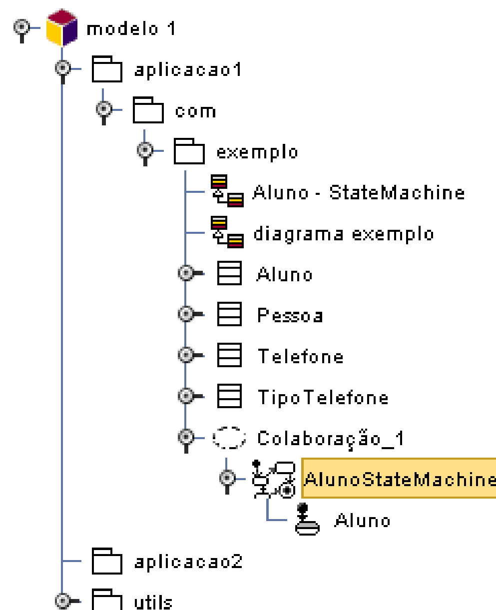


Figura 45 Criação da máquina de estados da entidade Aluno

O diagrama de estados deve possuir um único estado inicial com uma única transição para o próximo estado. Durante a geração o estado adjacente ao inicial é considerado o primeiro estado da máquina de estados. Assim, o estado inicial tem como objetivo indicar ao gerador qual será o primeiro estado. Na Figura 46 o primeiro estado é representado pelo estado ‘MATRICULADO’.

No framework Iguassu o diagrama de estados pode utilizar ações de entrada e saída. As ações de entrada são executadas quando a entidade troca de estados, assim que recebe o novo estado (onde esta a ação de entrada). As ações de saída são executadas antes de realizar a troca de estados, de um estado velho (onde está a ação de saída) para um novo. Essas ações são serviços *data services* (serviços gerados)ou *application services* (serviços implementados) que são executados durante a troca de estados, devendo possuir o nome completo do serviço que será executado. Na Figura 46 quando a entidade Aluno realiza a troca de estado de 'MATRICULADO' para 'TRANCADO'.Quando a entidade Aluno entra no estado TRANCADO, é realizado uma ação de entrada, ocorrendo a execução do serviço 'EnviarEmailCoordenadorAppService'.

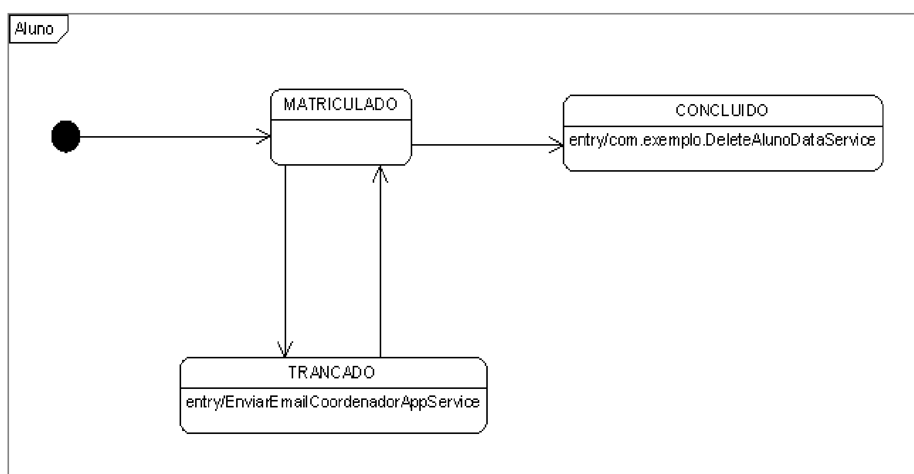


Figura 46 Máquina de estados da entidade Aluno

Assim, deve-se seguir procedimentos para modelagem dos diagramas de classes e estados permitindo que a geração seja realizada de acordo com as necessidades do framework Iguassu.

Por fim, os resultados foram a criação de dois geradores: CodeGen e BlueBox; e os procedimentos de modelagem. O primeiro utilizando tecnologia

XSLT, e o BlueBox utilizando o *templates Velocity* facilitando a programação dos mesmos. Os geradores necessitam que o arquiteto de software utilize os procedimentos para obter os dados corretamente do arquivo XMI.

5 CONCLUSÕES

Foram criadas duas ferramentas de geração automática de código: CodeGen e BlueBox as quais obtiveram ótimos resultados, visto que são capazes de gerar todos os códigos necessários à uma aplicação que utilize o framework Iguassu. Assim, descarta-se a necessidade de desenvolvedores dispensarem tempo com a implementação de serviços básicos, o que, otimizado, permite aos mesmos uma maior dedicação para com aspectos mais complexos do sistema.Como resultado, há, também, uma diminuição dos gastos da empresa durante a construção de software e facilitação da manutenção dos códigos utilizados.

Primeiramente fora criado o CodeGen e, com o acúmulo de conhecimento obtido, foi criado o BlueBox, que se mostrou superior ao CodeGen por ser uma ferramenta de fácil aprendizagem e manipulação de dados durante a programação dos *templates* mantendo, ainda assim, a qualidade dos resultados.

Também foram criados procedimentos para modelar uma aplicação Iguassu e que devem ser utilizados pelo arquiteto de software. Com a utilização dos mesmos houve melhoria na geração automática de código, evitando erros durante a ação do gerador.

Estes resultados foram implementados durante o estágio na empresa Mitah Technologies, conveniada à Universidade Federal de Lavras – MG, onde as ferramentas vem sendo aplicadas até o presente momento, o que possibilita um aprimoramento da produção desta empresa. Isto mostra a aplicabilidade e importância da existência da ferramenta em uma empresa de desenvolvimento de software que utiliza o framework Iguassu.

Após a implantação do CodeGen e os procedimentos de modelagem na empresa Mitah Technologies, o desenvolvimento das aplicações que utilizam o framework Iguassu obtiveram um ótimo avanço, concentrando os analistas de sistemas em pesquisas para evolução do Iguassu e no desenvolvimento de interfaces e serviços complexos para as aplicações. Com a dificuldade de manutenção dos *templates* XSL, foi proposto e desenvolvido o BlueBox, que permitiu uma rápida aprendizagem dos desenvolvedores com a programação dos *templates Velocity,* devido a legibilidade que os mesmos oferecem. Por fim, a Mitah Technologies tem adquirido ótimos resultados com a aplicação do gerador BlueBox e os procedimentos de modelagem, da mesma forma, é esperado para outras empresas que forem utilizar os mesmos.

5.1 Trabalhos Futuros

Como trabalhos futuros:

- Segundo GUTIERREZ (2009), é possível utilizar diagramas UML para geração de código de interfaces. Assim, a implementação de *templates* com o propósito de geração de interfaces aumentaria ainda mais a produção da Mitah Technologies bem como de qualquer empresa do ramo que utilizam o Iguassu.
- Utilizando-se como exemplo o GreenBox, uma perspectiva futura para este projeto seria a melhoria do BlueBox para que o mesmo passe utilizar como entrada não somente o arquivo XMI mas, também, o banco de dados, o que engrandeceria ainda mais a função desta ferramenta. Com isso, seria possível a utilização do BlueBox na manutenção de sistemas legados, sendo possível a transformação desses software em uma plataforma que utilize o Iguassu.

6 REFERÊNCIAS

APACHE STRUTS – Disponível em: http://struts.apache.org/ . Acessado em: 01 abr. 2010

ALEXANDRE, Douglas Barbosa. **Geração de Código Orientado a Serviço a partir de Modelos de Processos de Negócio**. 2009. 71f. Monografia (Curso de Ciência da Computação) – Universidade Federal de Lavras, Lavras, 2009.

CANOZA, Diogenes Corrêa**. Padrões de Projeto: Um Estudo de Caso Utilizando Um Framework de Desenvolvimento Orientado a Serviços**.2009. 63f.Monografia (Curso de Ciência da Computação) – Universidade Federal de Lavras, Lavras, 2009.

CASTRO, Lucas De Luca. **Procedimentos de modelagem visando geração de código com estudo de caso Java.** 2009. Iniciação Cîentifica Voluntária (Curso de Ciência da Computação) – Universidade Federal de Lavras, Lavras, 2009.

GreenBox – **Code Generator Tools**. Disponível em: https://greenbox.dev.java.net/. Acessado em: 11 mar. 2010.

GUTIERREZ, Felipe Gonçalves. **Geração Automática de Interfaces Gráficas a partir de diagramas de classes UML.** 2009. 74f. Monografia (Curso de Ciência da Computação) – Universidade Federal de Lavras, Lavras, 2009.

JAXB - **Java Architecture for XML Binding** .Disponível em: http://java.sun.com/developer/technicalArticles/WebServices/jaxb/. Acessado em: 16 fev. 2010

MCLAUGHIN B. **Java and XML Data Binding**. O'Reilly Media; 1 edition, 2002

MELLOR, S. J. et al. **MDA distilled: principles of Model-Driven Architecture**. Boston: Addison-Wesley, 2004.

NUCCIELLI, Renzo dos Santos. **Tratamento de Eventos em Java com Uso de Metadados**. 2008. 63f. Trabalho de Conclusão de Curso. (Graduação) – Instituto Tecnológico de Aeronática, São José dos Campos.

SAXON.**The XSLT and XQuery Processor**.Disponível em:http://saxon.sourceforge.net/. Acessado em: 16 fev. 2010

OBJECT MANAGEMENT GROUP. **MDA guide version 1.0.1**. OMG: 2003. Disponível em http://www.omg.org/docs/omg/03-06-01.pdf . Acessado em 22 mai. 2010

OBJECT MANAGEMENT GROUP. **MDA guide version 1.0.1**. OMG: 2003. Disponível em http://www.omg.org/docs/omg/03-06-01.pdf . Acessado em 22 mai. 2010

OMG – **Object Management Group**. Disponível em: http://www.omg.org/. Acessadoem: 17 abr. 2010

PANDA, D., RAHMAN, R. LANE, D.: **EJB 3 in Action.** Manning Publications, 1st edn.Abr 2007.

UML – **Unified Modeling Language**. Disponível em: http://www.uml.org/. Acessado em: 17 nov. 2009

XMI. **MOF 2.0, XMI Mapping Specification, V2.11**. Disponível em: http://www.omg.org/technology/documents/formal/xmi.htm. Acessado em: 12 jan. 2010

W3C (World Wide Web Consortium).Disponível em: http://www.w3.org/XML/. Acessado em: 20 fev. 2009

W3C - **Xml Path Language (Xpath) Version 1.0**, 1999.Disponível em: http://www.w3.org/tr/xpath/ . Acessado em: 10 mai. 2010

ANEXO 1 - TUTORIAL PARA UTILIZAÇÃO DO BLUEBOX

Introdução

Este tutorial tem como objetivo apresentar a utilização do BlueBox. O BlueBox é um gerador automático de código que tem como entrada uma base de dados que se encontra em um modelo UML. Este modelo é exportado para formato XMI (XML Metadata Interchange), o que permite o BlueBox através de um *parse* XML obter os dados que estão no modelo UML. Após receber os dados, o BlueBox utiliza modelos *Velocity* (*templates*) para definir como serão descritas as informações nos arquivos (código gerado).

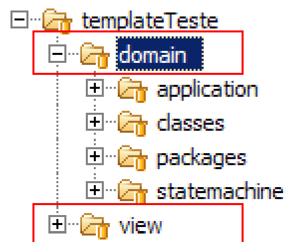
Download

svn checkout
http://www.mitahtech.dnsalias.net/mitahtech/trunk/BlueBox

Criando *Templates*

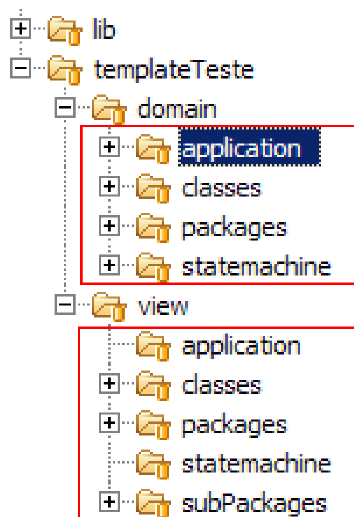
1 Criar um diretório onde serão criados os *templates*;

2 Criar uma pasta com o nome do pacote (no diagrama) do tipo do modelo;



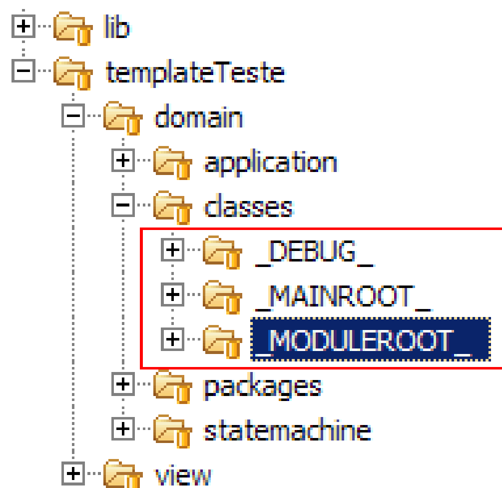
Em vermelho as pastas com os nomes dos tipos de modelos

3 Deve se definir em qual categoria será aplicado o *template* e criar (caso necessário) uma pasta com nome da categoria logo após o diretório do tipo do modelo;



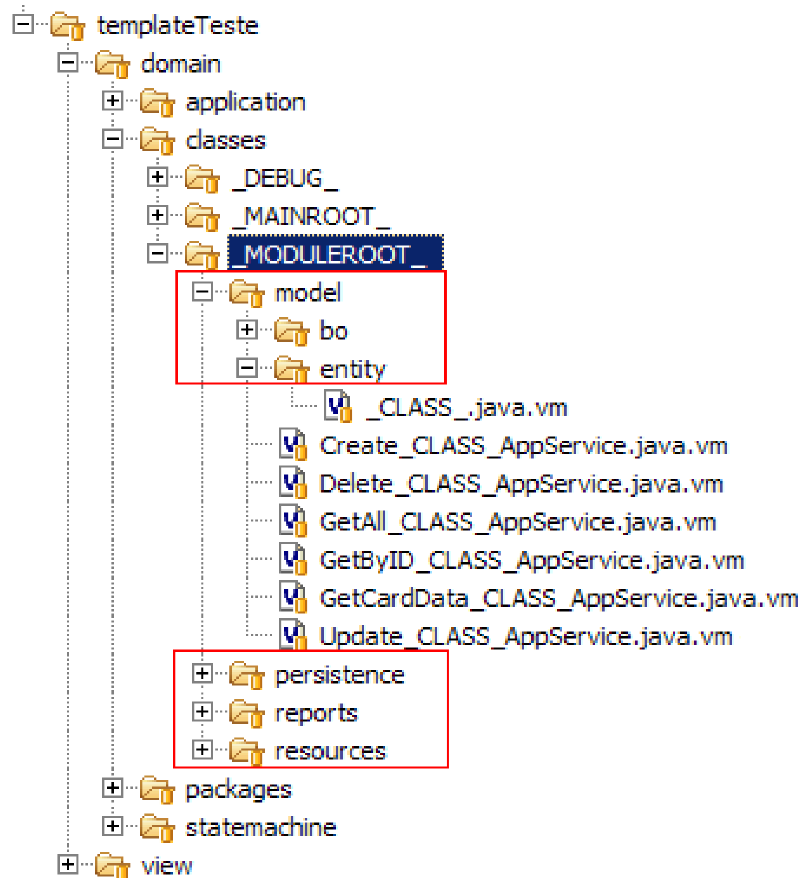
Em vermelho as categorias, selecionada a categoria application.

4 Definir onde será gerado o código (a partir da raiz do projeto) e criar uma pasta com nome da variável de ambiente.



Criado a pasta para variável _MODULEROOT_.

5 Caso necessário, criar pastas que serão os sub pacotes;



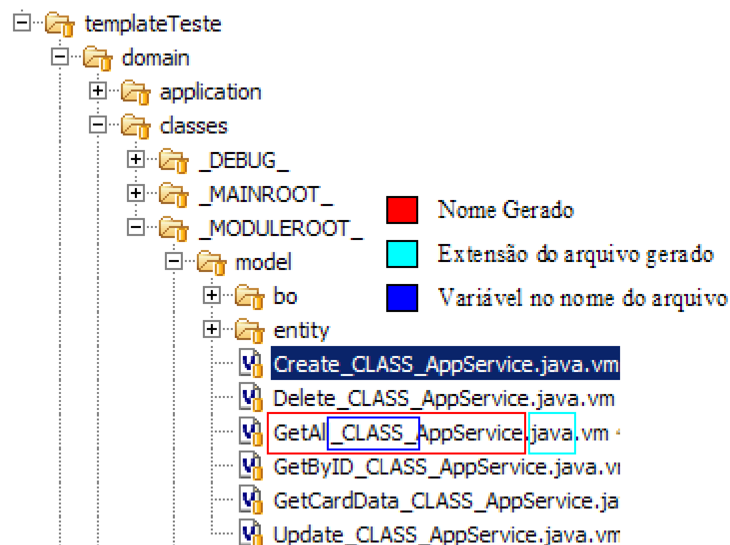
Em vermelho os sub pacotes.

1 Depois de definidas as categorias, variáveis e sub pacotes, cria-se se um arquivo que será o *template*. O nome do arquivo segue o seguinte formato: ' NOMEGERADO.EXTENSÃO.vm'

a. NOMEGERADO: será o nome do arquivo gerado. No nome podem ser colocadas algumas variáveis (com letra

maiúscula entre *underline* '_') que serão substituídas pelo seu valor durante a geração;

b. EXTENSÃO: extensão do arquivo gerado.



Criação do *template*

Programando

Nos *templates Velocity*, como já dito, é possível utilizar objetos Java durante a programação. Dentro da pasta docs, o projeto do BlueBox, possui o modelo UML do projeto BlueBox. No modelo encontram-se os diagramas de classes com as classes que representam os dados e possuem funcionalidades para serem utilizadas dentro do *templates*. Abaixo a localização dos diagramas:

- ✓ 'com.mitahtech.BlueBox.xmiParse.classDiagram': encontram-se as classes que representam o diagrama de classes no arquivo XMI;
- ✓ 'com.mitahtech.BlueBox.xmiParse.stateMachineDiagram': encontram-se as classes que representam o diagrama de maquina de estados no arquivo XMI;

- ✓ 'com.mitahtech.BlueBox.codegen.iguassuUtils': encontram-se as classes que possuem métodos que são úteis durante a programação;

Abaixo alguns casos comuns durante a programação dos *templates* que servem de base para a geração de código utilizando o BlueBox:

```
1  #foreach ($attribute in $class.attributes)
2      $attribute.visibility $attribute.name
3  #end
```

Listar todos os atributos de uma classe

```
1  #foreach ($attribute in $class.primitivesAttributes)
2      $attribute.visibility $attribute.name
3  #end
```

Listar todos os atributos primitivos de uma classe (que estão dentro da classe no diagrama de classes)

```
1  #foreach ($attribute in $class.associateAttributes)
2      $attribute.visibility $attribute.name
3  #end
```

Listar todos os atributos de uma classe que os tipos são de classes associadas(atributos do tipo BO

```
1  #foreach ($attribute in $class.attributes)
2      #foreach ($tagValue in $attribute.taggedValues )
3          #if ( $tagValue.name == 'X' )
4              #if ($tagValue.value == 'A')
5                  Tag: $tagValue.name com valor: $tagValue.value
6              #end
7          #end
8      #end
9  #end
```

Verificar se um atributo possui um marcador de nome X com o valor A

Executando o BlueBox

MainBlueBox.java é a classe principal que realiza a execução do BlueBox. Antes de executá-la é necessário definir os valores dos seguintes atributos:

outDir: caminho do diretório onde será gerado o código;

Ex: *outDir* = 'C:\\lucas\\workspace\\laborapix';

templatePath: caminho do diretório que contém os *templates*;

Ex: *templatePath* = '*template*Teste'; (quando esta na pasta raiz do projeto)

Ex: *templatePath* = "C:*template*Teste"; (encontra-se em outro local)

xmiFile: caminho do arquivo XMI;

Ex: *xmiFile* = "c:\\arquivosXMI\\laborapix.xmi";

Filtrar Classes:

Utiliza-se o método 'addFilteredClass(String className);' da classe *Velocity*Processor para adicionar as classes que serão filtradas durante a execução. Somente as classes que foram adicionadas serão geradas.

Ex: vl.addFilteredClass("Produto");

Somente a classe Produto será gerada

Filtrar Categoria:

Utiliza-se o metodo 'addFilteredCategory(String categoryName);' da classe *Velocity*Processor para adicionar as categorias que serão filtradas durante

a execução. Somentes as categorias que forem adicionadas serão geradas. Caso não seja adicionada alguma categoria, todas serão geradas.

Ex: vl.addFilteredCategory("classes");

Será gerado apenas os *templates* da categoria classes.

Filtrar *Template*:

Para filtrar um *template* deve-se utilizar o método 'addFilteredTemplate(String categoryName, String *templateName*);' da classe *Velocity*Processor. Somente os *templates* adicionados serão gerados. Caso não seja adiconado nenhum *template*, será gerado todos os *templates* daquela categoria.

Ex: vl.addFilteredTemplate("classes", "_CLASS_.java.vm");